UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

LipLink
Un cititor pe buze construit cu ajutorul învățării automate

Vlad Zincă

Coordonator științific:

Conf. dr. ing. Andrei Olaru

BUCUREȘTI
2024

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

LipLink
A Lip Reader built using Machine Learning

Vlad Zincă

Thesis advisor:

Conf. dr. ing. Andrei Olaru

BUCHAREST
2024

CONTENTS

SINOPSIS

Cititul pe buze reprezintă abilitatea de a extrage ceea ce se spune doar din urmărirea mișcărilor buzelor. Scopul lucrării curente este prezentarea succintă a lui LipLink, un program bazat pe tehnici de învățare automată al cărui scop este cititul pe buze, precum și a noțiunilor teoretice de bază din domeniul învățării automate ce stau la baza creării acestuia. LipLink este o rețea neurală complexă construită cu PyTorch ce reușește să generalizeze mimica buzelor prezentă în videoclipurile din setul de date GRID cu o acuratețe de 88,04%. Lucrarea prezintă problemele care au apărut în timpul dezvoltării lui LipLink, precum și soluțiile găsite.

ABSTRACT

Lip reading is the ability to extract what is being said based solely on the movement of the lips. The purpose of the current paper is to present LipLink, a program which uses machine learning techniques to lip read, as well as the theoretical notions that were required for its creation. LipLink is a complex neural network model built with PyTorch that succeeds in generalizing the mimic of the lips present in videos from the GRID dataset with an accuracy of 88.04%. The paper further presents problems that were encountered during the development of LipLink, as well as ways to solve them.

ACKNOWLEDGMENTS

Mădălina Savu, **Iulian Palade**, as well as the rest of the **Tobii AB Bucharest team**, for helping me with the development of the model and giving me valuable advice.

Conf. dr. ing. **Andrei Olaru**, for supporting my individual research and allowing me to pursue what I am passionate about.

My partner, **Alberta Milicu**, who supported me when the model was not working as expected.

My mother.

1 INTRODUCTION

1.1 Context

As Machine Learning has been expanding its scope more and more in the last few decades, new kinds of everyday problems begin to benefit from it. This is especially relevant to areas concerning human accessibility, with these new techniques allowing for great improvements.

The research to produce solid lip reading models has only been catching up in the last decade, with a major breakthrough being achieved Yannis M. Assael's LipNet [1] in 2016.

Since then, better and more complex lip reading models have been produced, however they often fall short of simplicity and are not very feasible in the real world.

1.2 Problem

Lip reading is the ability of understanding what is being said only by watching the movement of the lips. It is a dauntingly difficult task for humans, professional human lip readers only achieving an accuracy of $17 \pm 12\%$ [1].

It there follows that it would be a great achievement to build a program that can lip read with a good accuracy. This is the problem we are trying to solve: given a short video of a speaking person, we want to build a program that can output the discourse content as text.

We aim to do this by analyzing state-of-the-art research in this area and producing a Python-based program that implements the studied architecture.

1.3 Objectives

The main goal of the project is to produce a lip reader using machine learning techniques. In time, such a system could set the basis for a more complex system that could be mass-produced into smart glasses or similar devices and greatly improve the lives of hearing-impaired

people, especially in noisy environments where speech-to-text often perform poorly.

To be considered valid, this system would need to have a reasonable accuracy on a reputable dataset, at least significantly better than what seasoned human lip readers would get.

In engineering such a system, the author expects that valuable practical experience could be gained, that would greatly enrich the author's ability to tackle similar problems in the future.

1.4 Solution

The author proposes a three-step approach, namely to study state-of-the-art model architectures and similar research that has been done on the topic, attempt to implement a model using the knowledge that has been gained, and falling back to the first step when one finds itself in a stalemate.

After applying this approach several times, the end product is LipLink, a complex neural network system containing convolutional, recurrent and fully-connected layers.

1.5 Results

LipLink, inspired by LipNet [1], follows in the tracks of the former and is able to generalize the movement of the lips from the videos it learned from, obtaining an accuracy of 88.04%, all while simplifying LipNet's approach and using a smaller amount of resources.

The author's results, while being similar yet less computationally intensive than LipNet's, could be widely replicated, even on a personal computer, and could more easily serve as both a basis for smart device integration to assist hearing-impaired people, as well as a great introductory piece for amateur machine learning students who want to delve deeper into LipNet and other more complex lip reading models.

1.6 Structure

The paper follows a canonical 6 chapter structure and delves deeply into everything that is LipLink, as well as everything that was needed to achieve it. **Background** deals with the relevance of lip reading in the machine learning field as is today. **State of the Art** deals with the current available lip reading solutions, as well as their advantages and shortcomings. It then makes a compelling argument as to why the design choices for LipLink were chosen as

they were. **Solution** swiftly follows the steps that were taken to build the model and explains everything that was needed for its realization, while **Design and Implementation** delves deeper into the intricacies of bringing such an ambitious project to life. **Results** details the results that were obtained, as well as the metholodology of how these results were collected. Lastly, **Conclusion** sums up everything that was discussed and proposes subsequent research path that could be taken to enrich the field of machine learning-based lip reading.

2 BACKGROUND

Developing a lip reading algorithm based on machine learning techniques was an *obvious choice* considering the potential technical applications, as well as the growing interest in the field.

During the last few decades, an increase in computing power, coupled with an explosion of available data generated and the popularization of machine learning techniques such as deep learning [6] made many artificial intelligence-based solutions feasible for the first time.

This made many ideas that were once taught to be pure science-fiction come to life. And one of the fields that have felt this impact the most is computer vision.

The idea of using machine learning solutions to explore imagery and detect the patterns present is nothing new. Scientist Richard Szeliski[1] alleges that the idea of computer object detection was often discussed at prestigious American universities during the late 1960s.

2.1 MNIST

One computer vision problem that deserves discussion is classifying the MNIST database. The MNIST dataset is a collection of handwritten digits containing up to 60,000 training samples. Nowadays, training a model to classify this database with a reasonable accuracy is a popular task that features on most machine learning courses. Figure 1 shows some images from the MNIST database. Martin Kersting[2] alleges the reason for this is that *handwriting is the most difficult kind of character recognition*, and a computer program being able to do this task successfully would therefore prove beyond a reasonable doubt to any trainee that computers can be used for image object detection.

However, the way classifying the MNIST dataset is done has changed throughout time. The creators of the dataset, LeCun et al., which keep a detailed record of the major solutions to the problem [5], have been able to achieve a test error rate of just 12.0% in 1998 using just a simple linear classifier. Since then, the most notable breakthroughs have been in 2004,

[1] Szeliski, Richard. Computer Vision: Algorithms and Applications. Springer Science & Business Media, Springer, New York, 2010

[2] Kersting, Martin. Support vector machines speed pattern recognition. https://www.vision-systems.com/print/content/16737424

[3] © https://commons.wikimedia.org/wiki/File:MnistExamplesModified.png

Figure 1: Images from the MNIST dataset[3]

achieving a 0.42% error rate with a three-neuron classifier called LIRA, in 2011, when using random distortions yielded just 0.27%, 0.25% in 2016 using convolutional neural networks, and lastly 0.18% in 2018 by Kowsari et al.[4] using a model containing convolutional, recurrent and fully-connected neural networks at the same time. Figure 2 shows the progress made in classifying MNIST the last two decades.

This evolution, both in the involved methods as well as in the obtained results, lends us a good lens for understanding the current scientific situation in the field of computer vision and walks us through the evolution of machine learning itself. This understanding will prove *crucial* later on.

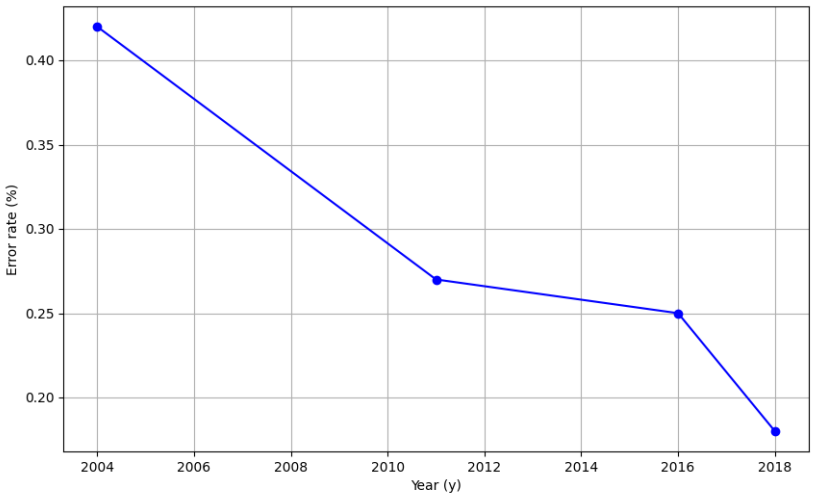

Figure 2: Error-rate evolution of solutions that classify the MNIST database. Data given by LeCun [5]

[4]Kowsari, Kamran et al. RMDL: Random Multimodel Deep Learning for Classification. ICISDM '18: Proceedings of the 2nd International Conference on Information System and Data Mining, 19-28, 2018

2.2 Motivation

The creation of better and more refined machine learning models and results is coupled with a significant growth in the interest of the general population in the field. According to key research done by McKinsey & Company [6], the Bureau of Labor Statistics projects a 23% growth of the number of machine learning engineering positions in the US between 2022 and 2032.

This is why the author has come to the conclusion that the current field of machine learning finds itself in an antithesis that could prove very hard to solve. As exemplified by handwriting classification, the development of models that provide smaller and smaller error rates demand the use of more and more sophisticated and complex neural network architectures. This is contrasted by the popularization of machine learning in recent years, with more and more enthusiasts being drawn into the field than ever. The point is that this contrast may create a *gap of understanding*, such that many people studying machine learning for the first time today would have trouble understanding the more sophisticated state-of-the-art papers getting published today.

In a field that is notorious for being incredibly complex and hard to grasp, one of the main pillars of motivation for the author has been the *development of simplified approaches* to solving complex problems, that still prove feasible and are so much easier to understand, especially for amateurs that are studying machine learning for the first time.

Another pillar of the author's motivation has been the expanding of the scope of machine learning solutions to areas that have *not* been thoroughly researched. Common problems that are being solved with machine learning, such as handwriting, face or street sign classification, were considered unsuitable for the present paper due to the wide popularity of the topics, and the inexhaustible research that comes with them.

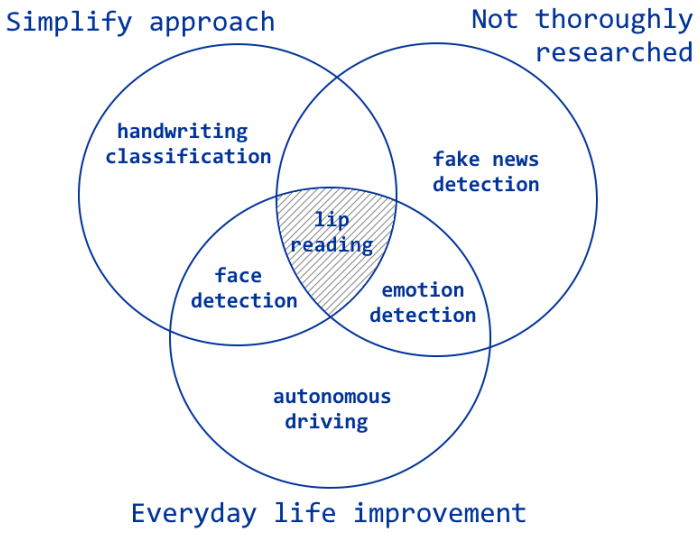

Figure 3: Venn diagram showing the motivation for different topics considered for research

On top of all that was discussed, problems of interest were those whose breakthrough and popularization would mean a great improvement of everyday life and well-being for a significant amount of people, as the author believes it to be the ultimate goal of the advancement of computing and, on a broader sense, technology.

2.3 Lip reading

In this context, the problem that became the front-runner for this paper became **lip reading** (see Figure 3). For the unfamiliarized reader, lip reading is the ability of using just the movement of the lips to determine what is being said. For most of human history, it has mainly been used by people with limited hearing ability to discern what their peers are saying. Researchers Lynne E. Bernstein et al.[5] allege that, in the last century, there have appeared a wide range of schools and materials teaching people how to improve their lip reading ability, for reasons ranging from better integrating people with limited hearing ability into our society to even espionage.

Delving deeper into the matter, it is easy to see how lip reading fits our research topic criteria.

Firstly, compared to handwriting classification, emotion detection or driving, humans ability to lip read are notoriously poor, with researchers Yannis M. Assael et al. [1] estimating that human hearing-impaired lip readers achieve a performance of only $17 \pm 12\%$. There it follows that, compared to the other detection and classification human tasks that are in line for machine automation, there should be some urgency into automating lip reading and other tasks that humans cannot consistently do.

This is the main reason while developing a computing system that can consistently predict what is being said based only on the movement of the lips would be a great improvement in the lives of many people: compared to the other tasks that we considered for this research, lip reading is by far the one that human accuracy is remarkably low.

We envision the main purpose of such a system to be the help it would lend to people suffering from hearing loss. While a system that uses the sound instead of the lip movement is a very good solution to this societal problem, we envision that the two systems, one sound-to-speech and another *lip-to-speech*, could be integrated together and would yield better results, especially in noisy environment where the accuracy of just a sound-to-speech system could be greatly impacted.

For ease of use, we could imagine that future interdisciplinary research into the field of machine learning and low-power computing could produce some kind of glasses-like device containing a small system-on-a-chip to empower speech detection and return the results as a sort of

[5]Bernstein, Lynne et al. Lipreading: A Review of Its Continuing Importance for Speech Recognition With an Acquired Hearing Loss and Possibilities for Effective Training. Am J Audiol, 31(2):453-469, 2022

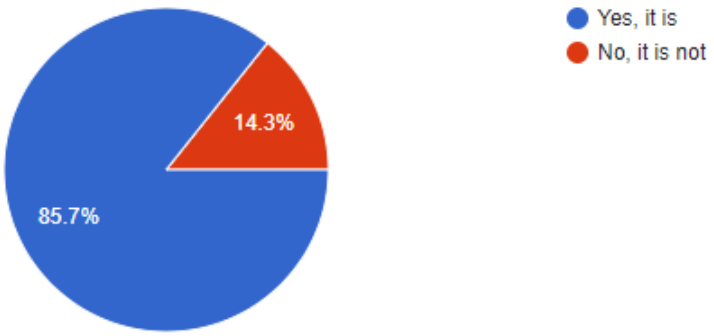

Figure 4: Chart depicting the results for question *Is it a problem that our society does not facilitate better accessibility for hearing-impaired, including deaf people?*

subtitles to the hearing-impaired person's glasses. This solidifies the need for a simplified approach to the process of lip reading that this paper tries to capture.

However, the author does realize that the thought offered above is pure conjecture, and the technology of the future could end up massively different.

Secondly, while the importance of research for developing a machine learning-based algorithm for lip reading has been established formerly, the research in the field has not been enough, with the seminal paper on the matter being produced by Yannis M. Assael et al. [1] only in 2016. While systems for detecting lip patterns have got better and better over the years, they also increasingly grew in complexity, which radically differs from the stated goal of the author of laying the basis for a simplified system.

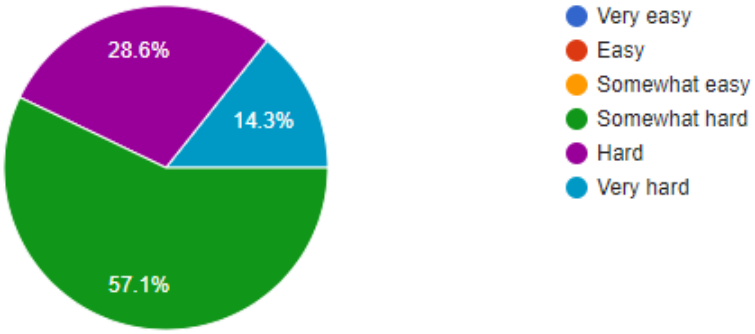

Figure 5: Chart depicting the results for question *How hard do you believe it is for the average person to study and understand machine learning?*

As established, developing a simplified machine learning solution for lip reading is a *technical necessity* that would open the door to a wide range of technical application that could greatly benefit human life, especially for hearing-impaired people.

The author's motivation is corroborated by a survey[6] that they created in order to better understand the societal perception of such a work. In survey *Importance of accessibility for hearing-impaired people*, questions about human accessibility, as well as the difficulty of understanding machine learning were asked, and the answers that are provided in figures 4 and 5 support the notion that a simplified approach to automated lip reading could greatly benefit society.

[6]Results are taken from survey https://docs.google.com/forms/d/1TVSSd-6J0jNfakCveguvBmEiN5yhvqiAqM9M4qzvZYs/

3 STATE OF THE ART

The academic, structurally-cohesive study of lip reading only started during the mid-20th century, when academics started to notice how mouth perception during speech plays a crucial role in human communication and even sound perception. It was not long until scientists started to figure that following the lips during speech plays an integral part into understanding what is being conveyed, and that speech is perceived by the human brain not only as an auditory information, but also as visual.

This has beeen dubbed the McGurk effect, as it has been popularized by scientist Harry McGurk[1] in the 1970s.

Since then, linguists spent the following decades attempting to better understand the movements of the lips and how it plays a part in decoding what is being said. Intuitively, it is known that different sounds are exhibited differently because they are caused by different movements of the lips, mouth and esophagus.

Figure 6: Image showcasing different visemes[2]

However, it was later established, especially in the research of Fischer[3], that this understanding will not be enough to solve the intricacies of lip reading, as it is much more complicated than that. For example, Fischer proved that the two different sounds can look the exact same when considering the lips, mouth and face. This created the concept of *viseme*, an equivalent of

[1] McGurk, Harry and MacDonald, John. Hearing lips and seeing voices. Nature, 264:746-748, 1976

[2] © https://www.researchgate.net/figure/Example-visemes-for-phoneme-classes-given-in-Table-1

phoneme into the visual space. Figure 6 shows different human visemes. It is further alleged that all the visemes in a language, meaning all the movements that the lips could make to produce all the sounds, can be mapped to the different phonemes, or sounds, in a language, but not in a bijective, one-to-one manner that researchers would have liked.

The research continued slowly in the 1980s and 1990s, when the idea of automated lip reading started to gain popularity due to the rise of science fiction in pop culture. In the 2000's, the idea was taken by artificial intelligence researchers which, using techniques such as Hidden Markov Models (HMMs) and Support Vector Models (SVMs), attempted to implement lip reading algorithms with varying degrees of success.

3.1 GRID

One major breakthrough was achieved in 2006, with the creation of the GRID dataset by Cooke et al. [3]. Even though there were many other datasets that could be used before that, Yannis M. Assael [1] claims that the GRID corpus was the first that was not focused on capturing only single words, but actual sentences. This is extremely beneficial for two reasons: (1) because it allows us to see how existing lip readers would perform on real-life phrases and (2) because it gives us *context*.

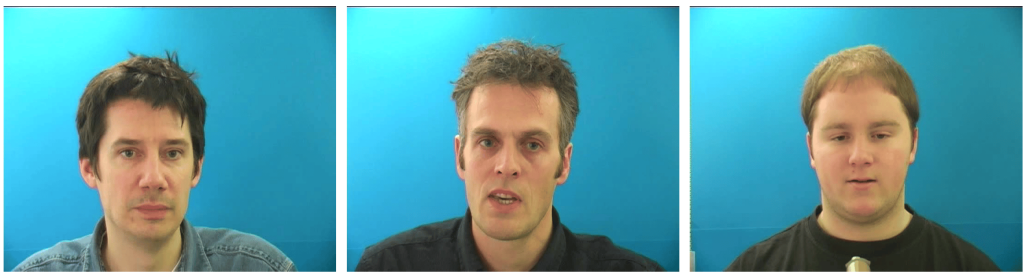

Figure 7: Example frames from the GRID dataset showing different speakers[4]

With lip reading, context is very important, even in human trials. When the human lip reader knows the context another human is discussing about, there is a stark increase in its accuracy, with 10%. [1] This is because, even though many visemes could look the same and that would result in several words being the possible accurate prediction, with context the human can deduce which one of these words was actually used.

This is why, if we want to achieve good accuracy with machine learning-based lip reading, it follows that we would need to somehow teach the model about this language context.

[3]Fischer, C. G. Confusions among visually perceived consonants. Journal of Speech, Language, and Hearing Research, 11(4):796-804, 1968

[4]© https://www.mdpi.com/2076-3417/11/15/6975

GRID is structured as follows: there are videos of 75 frames each of a speaker producing a sentence, in *.mpg* format. For reference, one can check Figure 7. Each video corresponds to an alignment file, with the same name but stored as *.align*, that contains the words that the video contains, as well as the moments, given in miliseconds, in which every word starts and ends. No word being said is marked by the word *sil*. Listing 3.1 showcases an alignment file example.

Another advantage of the GRID dataset is the sheer size of it: according to the database's creators [3], it contains 34 speakers producing 1000 sentences each, with a total of a staggering 28 hours. This is why, since its inception, the GRID dataset has been the standard for lip reading benchmarking, however, as one can imagine, progress in the 2000s was slow.

```
0 23750 sil
23750 29500 bin
29500 34000 blue
34000 35500 at
35500 41000 f
41000 47250 two
47250 53000 now
53000 74500 sil
```

Listing 3.1: Example of how an alignment is codified in the GRID dataset. The speaker in the corresponding video is saying "bin blue at f two now"

With the popularization of deep learning techniques beginning around 2012 with the emergence of AlexNet, there was a renewed interest in the field of lip reading, both from reputed scientists and the public at large. This growing interest materialized itself in the 2016 work [1] of Yannis M. Assael, Brendan Shillingford, Shimon Whiteson and Nando de Freitas, *LipNet: End-to-End Sentence-level Lipreading*.

3.2 LipNet

If the emergence of the GRID dataset provided the world for the first time with the data necessary to train an automated lip reader, LipNet went out and truly gave us a basis to solving it.

The explosion of deep learning popularity eventually led to its application in the field of automated *sentence-level* lip reading, however at first it was considered a challenging task to solve due to the complexity of the data involved: when lip reading, both the spatial - the position of the lips, and temporal - the passing from one position to the other over time are crucial in making it work.

The sentence-level bit is crucial: at the time LipNet was published, the focus was mostly on *word-level* lip reading, for which older techniques lent satisfying results. The need, however, in real life is for models that can predict complex, longer phrases that people actually do communicate. This is what made the author look for sentence-level solutions, rather than improving the already-existing word-level ones.

And as discussed in chapter 2, 2016 was a time of experiment: it was the first time when a complex neural network containing convolutional, recurrent and fully-connected layers was applied to produce a model that would get a 99.75% accuracy rate on the MNIST dataset.

A similar solution was provided by the LipNet's creators as well: the mouth regions were cropped from the frames, and they were fed into a neural network consisting of spatiotemporal convolutional layers, bi-directional GRUs (RNNs) and a fully-connected layer, which produced an output that was compared with the alignments using a special CTC loss on the basis of which the weights are being updated. Figure 8 is a visual representation of this architecture.

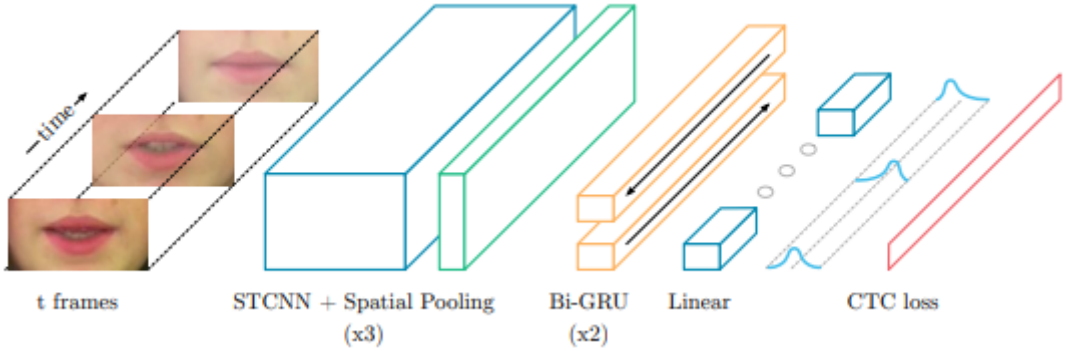

Figure 8: The architecture used by Assael et al. to implement LipNet[5]

Note that all these notions may seem complicated to a beginner, and the understanding of them that the author has gained in order to implement a similar system is detailed in the following sections.

However, this complex approach is the recipe to success for solving automated lip reading, at least from the deep learning theoretical framework that has taken root in machine learning circles in the last decade.

The authors of the paper even provide us with saliency maps, such as those seen in figure 9, of the backpropagation for some key words, highlighting how the model is focusing on the correct areas of the input image for every *viseme*.

LipNet was the first to provide us with a complete, sentence-level automated lip reading solution. According to its creators [1], LipNet achieves a 95.2% accuracy and just 4.6% WER (Word Error Rate) on the GRID dataset, which at the time was much more than the existing word-level models, and cemented its place as the seminal paper on the issue of automated lip

[5]© Yanis M. Assael et al. https://arxiv.org/pdf/1611.01599
[6]© Yanis M. Assael et al. https://arxiv.org/pdf/1611.01599

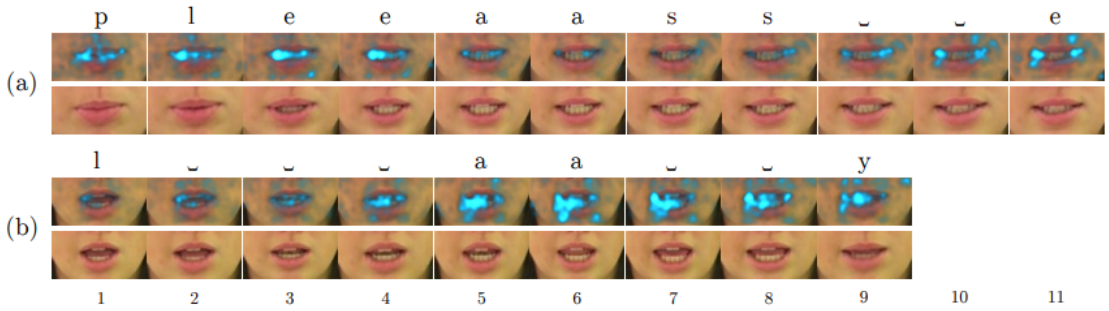

Figure 9: Saliency maps for words "please" and "lay", as provided by Assael et al.[6]

reading: every model that came after LipNet has learned and been influenced by it in some form or another.

Assael et al. provided full implementations of LipNet in both PyTorch and TensorFlow, which are the two most popular Python-based machine learning frameworks that will be discussed shortly. However, these implementations are rather complex, especially considering the background of the author.

3.3 LCANet

After the success of LipNet, there appeared many other state-of-the-art models that obtain even better results, all while complicating the model further.

One such model is LCANet, developed in 2018 by a team of American researchers consisting of Kai Xu, Dawei Li, Nick Cassimatis and Xiaolong Wang. In the paper concerning the development of LCANet [2], they claim that LCANet builds on the previous work on LipNet's creators mainly by (1) incorporating a highway network between the STCNN and GRU layers that allows the network to pass information to the output directly and (2) by using an Attention mechanism buitl on top of the CTC loss function that captures context and permits the model to learn to focus on the most important parts of the input.

Xu et al. [2] mention that LCANet's word error rate (WER) stands at 3.0%, which is a significant improvement from LipNet. The paper has played an important role in corroborating the idea that Attention mechanism should play an important role in improving the already-existing lip reading methods in the field of deep neural networks.

3.4 Pingchuan's VSR

At the time of writing, the model that performs best on the GRID dataset is described by Pingchuan et al. [4] in their 2022 work *Visual Speech Recognition for Multiple Languages in the Wild*. Their work provides us with a model that outperforms models that use up to 21 times more data and that can even be improved by training on videos with self-generated transcripts. For this, they use several techniques that are outlined below.

One of the most important features that the authors stress about their work is time-masking. Time-masking basically consists of deliberately removing similar frames from a video so the model is forced to place more emphasis on a singular frame and the lips' position in it. According to the authors [4], this method has been used by audio-to-text machine learning models for a long time.

Another factor that leads to the phenomenal results of Pingchuan's VSR model is the emphasis placed on optimizing the model's hyperparameters. By 2022, most people have understood that the use of both convolutional layers for spatiotemporal feature extraction, as well as recurrent layers such as LSTM or GRU for temporal modelling, are paramount, however the numbers and sizes of these layers, and the learning and dropout rates can also play a significant role, according to the VSR model's creators. [4]

Based on the aforementioned observation, their architecture modifies LipNet by: (1) adding a ResNet-18 layer in-between convolutional and recurrent layers that, similar to LCANet's highway network, aids in deeper feature extraction, (2) uses up to 12 layers of RNN layers and (3) similar to LCANet, uses a hybrid CTC/Attention loss.

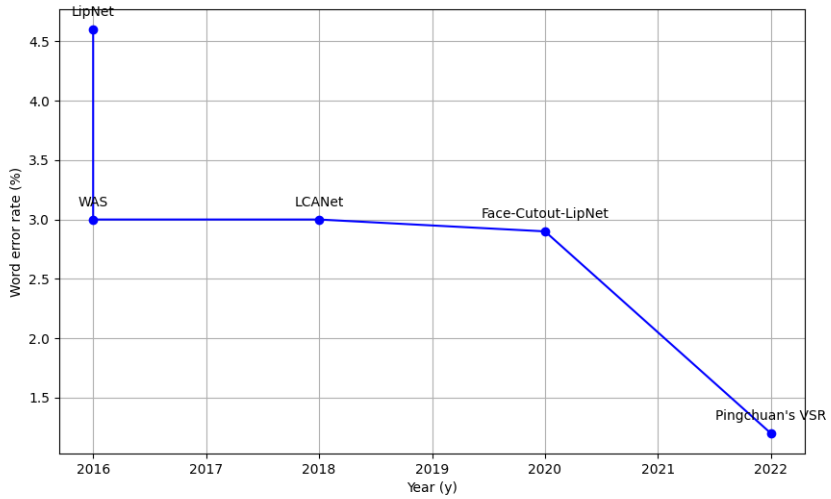

Figure 10: Word error rate (WER) evolution of state-of-the-art lip reading models inspired by an online[7] GRID benchmark

[7]Lipreading on GRID corpus (mixed-speech). https://paperswithcode.com/sota/lipreading-on-grid-corpus-

According to the author's research, Pingchuan's Visual Speech Recognition (VSR) model is incredibly sophisticated and constitutes the current state-of-the-art approach to lip reading, at the time of writing, as exemplified in figure 10. The model achieves a staggering 1.2% word error rate on the GRID corpus, however Pingchuan et al. [4] note that further developments in the field of automated lip reading can be achieved by using a larger and more generalized dataset.

3.5 Lip Reading in the Wild

The current paper's author also considered using other datasets than the classical GRID, made by Cooke et al. [3] all the way back in 2006, especially when taking into account the advice given by Pingchuan et al. [4], as well as their proof that diversifying the dataset and even providing videos with automatically-generated text could increase accuracy.

One of the main contenders of GRID was developed by Joon Son Chung and Andrew Zisserman[8] in 2017 and is called Lip Reading in the Wild (LRW). The team of two researchers created a program that can easily generate a dataset using TV transmissions, and they further used it to make the LRW dataset. It features 500 different words being spoken 1000 times each by more than 100 different speakers.

The dataset is quite diverse and has gained popularity among automated lip reading researchers, including Pingchuan et al. that heavily rely on it.

Table 1: Comparative analysis of GRID and LRW databases

Metric	**GRID**	**Lip Reading in the Wild**
Number of speakers	34	>100
Number of unique words	around 50	500
Number of sentences	34,000	0
Frames per video	75	29
Environment	Laboratory	Real-world

While GRID provides data that is low diversity and obtained in a controlled environment, as can be observed from table 1, the data from LRW is much more dynamic, with various backgrounds and lighting. This is why scientists that want to build a lip reading model that functions well in real-life settings, such as Pingchuan, have started using LRW and similar datasets in the detriment of GRID.

While the author supports the use of building more robust lip reading models, they have ultimately chosen *against* the use of Lip Reading in the Wild for this paper, due to two

mixed-speech

[8]Chung, J. and Zisserman, A. Lip Reading in the Wild. Computer Vision – ACCV 2016, 10112:87-103, 2017

reasons. Firstly, videos in LRW are focused on a single word each, and this stands contrary to the breakthrough made by Yannis M. Assael et al. [1] that produced a sentence-level lip reader. Even if the current paper used LRW for training, one would still need to test sentence-level accuracy on GRID or a similar dataset. Secondly, while using a dataset such as LRW can be effective in decreasing the word error rate of a lip reading model, it significantly increases the complexity of our program and training and this stands opposite to this paper's stated goal. The author does not want to produce a complicated, state-of-the-art lip reader, but rather tackle the problem using a simplified, easy to understand approach that still achieves a good accuracy.

It is with this in mind that the author decided to use the GRID dataset and base their work around Yannis M. Assael's [1] LipNet model architecture, the simplest sentence-level lip reader, and look to provide an easier to grasp approach that is still able to lip read consistently.

3.6 Renotte's LipNet

The search for a simplified approach to LipNet has attracted the author's attention to Nicholas Renotte's work. Nicholas Renotte is an artificial intelligence engineer from Sydney, Australia, that makes videos on YouTube providing simple machine learning solutions. At the time of writing, their videos[9] have gathered more than 17 million views.

One of Renotte's video, *Build a Deep Learning Model that can LIP READ using Python and TensorFlow*, provides an unorthodox, yet clever approach to lip reading that has deeply influenced the current work's author. In it, Renotte [9] uses TensorFlow to build a model similar to LipNet consisting of 3 convolutional layers, 2 LSTM layers and 1 fully-connected layer. While only using 1 speaker from the GRID dataset, the CTC loss still converges to around *0.1* and the model can predict with a large degree of accuracy most of the sentences reserved for testing.

The model that Renotte has produced features a certain level of simplicity that aligns with the paper's stated goal. This simplicity is precisely why Nicholas Renotte's work constitutes one of the main influences of this paper's proposed solution.

3.7 Python

With the author looking to build upon LipNet's sentence-level model and Nicholas Renotte's simplified approach, the only choice that remains to be made is the tools, such as programming

[9]Nicholas Renotte - YouTube. https://www.youtube.com/@NicholasRenotte

language and framework. For this, the main candidate has been Python from the beginning.

At the time of writing, Python needs no introduction. With 8.2 million users, Python is the world's most popular programming language, according to the TIOBE index[10]. It has been steadily rising in popularity in the last few years due to its simplicity and large collection of packages.

Python is an easy to learn, dynamically typed language that appeals to a lot of people. It is multi-paradigm, but has taken a lot of features from functional programming, the proponents of which place a great deal of importance on clean code and *immutability*.

Aside from that, Python has become an obvious choice due to its large collection of packages, including some of the most popular frameworks for machine learning in use today, as well as a large community that makes a forum readily available on the Internet for any imaginable problem.

It is in this context that Python became LipLink's language of choice in the detriment of languages like *R*, which is popular among data science and machine learning researchers but cannot really compete with Python for non-specific Machine Learning tasks.

3.8 PyTorch

For the choice of framework, there were three options available: PyTorch, TensorFlow and Caffe. Currently, the most popular option is PyTorch, a Python library developed by Meta.

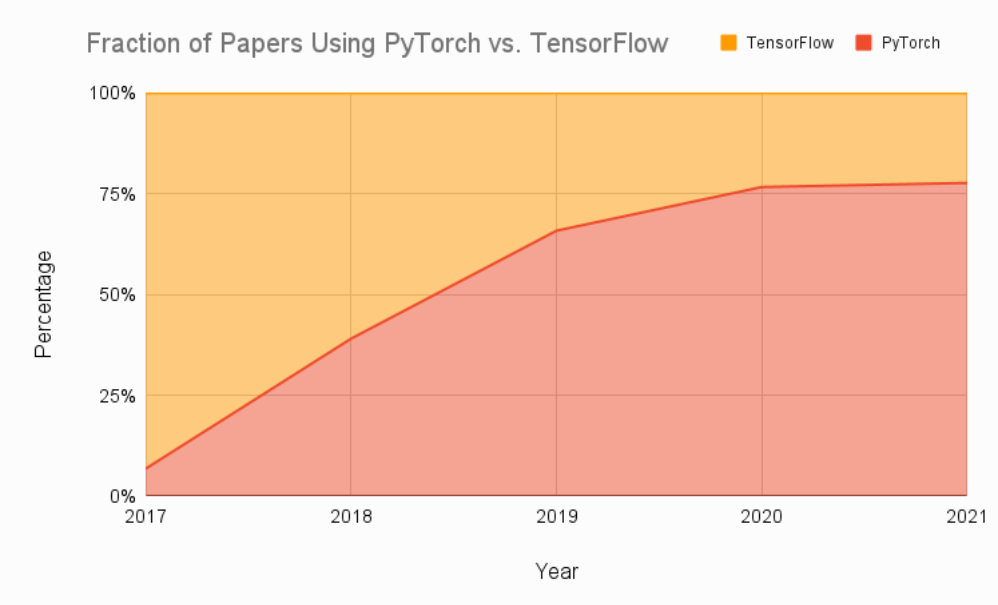

Figure 11: Graph[11] showing the increased popularity of PyTorch in research papers

[10] Jansen, Paul. TIOBE Index. https://www.tiobe.com/tiobe-index

PyTorch is well-known for its dynamic approach to neural networks, which makes it well-suited for research in the area. The code structure of PyTorch development is more closely related to Python synthax, thus making it attractive for people who have prior knowledge of Python.

Another advantage to utilizing PyTorch is the larger and more clearly defined documentation. While TensorFlow evolved to cater specifically to engineering and deployment purposes, PyTorch has been seen, since its inception, to be more oriented towards researchers, according to O'Connor [8], and as such has a more extensive and clear documentation.

On top of this, another major advantage to PyTorch is its current popularity, as exemplified by figure 11, which translates to well-maintained code base and a large community who have come across most of the PyTorch-specific problems that one could encounter.

All these advantages make PyTorch a strong candidate for the contender of Python-based machine learning framework to be used.

3.9 TensorFlow

A few years ago the most popular machine learning framework, TensorFlow, developed and maintained by Google, is a contender to PyTorch that is hard to overlook.

According to O'Connor [8], TensorFlow was created in 2015 to resolve the deployment and scalability problems that were associated with deep learning models, and the framework is still considered more extensive than PyTorch when it comes to these issues.

Moreover, Renotte's LipNet was written using TensorFlow [9], and following a similar implementation for LipLink would be easy if one were to use TensorFlow.

However, as good of an option it could be for specific problems, the author believes that TensorFlow is not as well suited as PyTorch for the development of LipLink, due to a number of reasons.

First of all, TensorFlow code is perceived [8] as more complicated and harder to debug than its competitor, and more oriented towards delivering scalable products, which is the opposite of the stated goal of LipLink, namely to provide a simplified, proof-of-concept lip reader.

Second of all, TensorFlow is considered, including by its experimented users such as Renotte [9], to have a much more limited documentation than PyTorch, and since a secondary goal to the current paper has been to obtain the valuable experience needed to solve similar problems in the future, using methods that are poorly documented would be the downright opposite.

Last but not least, it is a well-known fact [8] that TensorFlow has lost popularity in recent

[11] © https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/
[12] © Source of data is Google Trends

Figure 12: Graph showing the popularity of PyTorch and TensorFlow over time[12]

years. If trends continue, figure 12 shows how PyTorch will be increasingly dominating the field of machine learning for years to come, and writing LipLink with PyTorch would only contribute to the paper's credibility and relevance in the years to come.

This reasoning makes the use of TensorFlow unfeasible for the construction of LipLink, and PyTorch the much better alternative going forward.

3.10 Caffe

Caffe is an emerging, very fast deep learning framework developed by Berkeley that has been considered briefly for LipLink. Its main advantage is the ability to define neural network models using just text files. This would definitely have a large positive impact on experimentation speed.

The final choice has been made in favour of PyTorch due to the fact that Caffe has a very small community and the development of LipLink with it would significantly curb its reach, as shown with TensorFlow.

Nevertheless, the author considers Caffe a powerful framework that could have a significant increase in popularity in the coming years.

4 SOLUTION

Based on the author's experience, the development of most solutions that are based on deep learning models follows a strict *five step approach*:

- **Fetch data**, the step that leads to obtaining the raw dataset needed to train and test a neural network model,
- **Preprocess data**, during which the focus is modifying, standardizing and/or labeling the raw data so as to make it more suitable for feeding in a neural network,
- **Create and train model**, when one develops a model architecture and decides, following multiple experiments, the optimal architecture and hyperparameter configuration for best results,
- **Decode data**, when the results returned by the network are transformed so as to be human readable and easily interpreted, and
- **Test model**, during which one attempts to check the robustness and quality of the application.

One should however keep in mind that some of these steps can and will sometimes overlap and that there are a lot of intricacies regarding each step that have not been discussed.

The existence of these five steps, along with what their descriptions entail, lead us to a relevant conclusion. In order to develop a machine learning-based solution, creating and training a neural network model is just a small part of the puzzle, while most of the other steps of the application pipeline involve notions that have *less* to do with machine learning and more to do with general software development best practices.

As can be seen in figure 13, a similar approach to the above-mentioned one was used during the creation of LipLink as well, and this is why this section will focus *equally* on describing each step and its corresponding Python class structure, as well as the neural network architecture, highlighting the importance of every layer in the model.

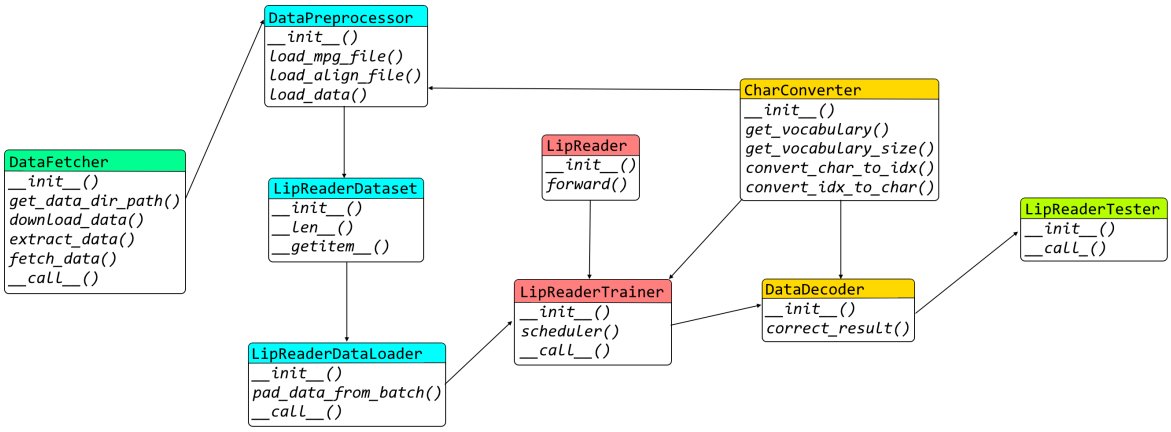

Figure 13: LipLink *application pipeline diagram*, showcasing every concrete class, their use in relation to other classes and their correspondence to the five steps approach, specified through colors

4.1 Fetch data

Fetching data represents the methods and processed used in machine learning to obtain the data needed to train the model and solve the desired problem.

Generally, it is the first step taken in solving a problem with machine learning. Contrary to the classical way of solving computing problems, when one devises a *correct* algorithm that takes in an input and applies a series of computational steps on it in order to obtain a solution the reception of which is *easily* explainable, machine learning takes a radically different approach.

To understand the difference between two, we could take the example of a classification problem, such as handwriting classification, that was described before. If one were to solve it programmatically, one would look at devising an algorithm that would look at the shapes in the image iteratively and compare them to the known shapes of the ten digits, admitting some kind of margin for pixel intensity to allow for better accuracy. For exactly ten known images of digits, the approach for checking the similarity between each of the ten image and the input image is most robust, and would work well enough.

The problem is that, in our day-to-day life, we do not want to just classify ten exact images of the digits anymore: writing digits is very different, and they are written today using a wide arrange of fonts, sizes and positions. When considering handwritten digits, the problem gets even more complex, because people write digits in very different ways sometimes.

Still, in some way or the other, for humans, the task of classifying digits is trivial. The approach machine learning takes on this problem is not to devise a very good iterative algorithm that could detect the digits, but rather, to simply ask *what it is that humans do*.

The short, simple answer to that question is that, in order to be able to classify the digits so well, humans, throughout their lives, *looked* at a lot of digit images. There is no other way to do it. So the way to solve classification problems, the machine learning way, is to simply devise a program that could *look* at a lot of data and *learn* from them.

But in order to look at a lot of data, one must *have* a lot of data. This is where data fetching comes into play.

For lip reading, the relevant data is formed of videos of people moving their lips to say something, and what they actually said, as text.

In order to get such data, one would need to:

- Create such data themselves, either alone, with other researchers or using the services of an organization specialized in data acquisition, or
- Obtain some data already collected by someone else, from the Internet.

For our short development cycle and stated research goals, the second method proved much more convenient. Due to the reasons already discussed, the dataset that will be used by our program is the GRID dataset.

With the goal of increasing simplicity and reproducibility, we have devised a Python class that fetches this automatically acquires the dataset from the Internet, and called it intuitively *DataFetcher*.

4.1.1 DataFetcher

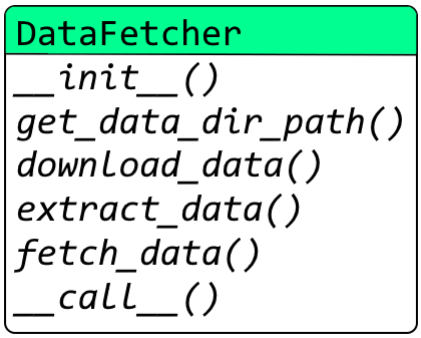

Figure 14: Diagram of *DataFetcher* class, taken from figure 13

The purpose of the **DataFetcher** class is to automatically obtain the data needed by the training process of LipLink in the current working directory from which it is called.

As observed in diagram 14, *DataFetcher* contains the following methods:

- `__init__(data_url, data_dir_name, data_file_name, data_file_size)`, which initializes an instance of the *DataFetcher* class,
- `get_data_dir_path(data_dir_name)`, which gets the path to the directory where to store the data, relative to the current working directory,
- `download_data()`, which downloads the GRID dataset as a `.zip` file,
- `extract_data()`, which extract the `.zip` file in the desired directory,
- `fetch_data()`, which calls the two aforementioned methods and does the entire process together, including removing the `.zip` file, and
- `__call__()`, which makes a *DataFetcher* object directly callable.

To download the data, the application uses the `requests` Python package in order to send a GET request to a webpage that contains the entire GRID dataset and attempts to download it.

Upon success, the resulting `.zip` file is unzipped using the `zipfile` Python package, after which the program makes sure to delete the remainding `.zip` file and notify the user that the process has ended.

The current design handles the case in which the dataset is not downloaded correctly from the webpage and reattempts the download until it finishes correctly. This handling is further detailed in chapter 5.

4.2 Preprocess data

After the data has been downloaded locally, one can start the process of modifying it accordingly so that it can be fed into the neural network model.

This is one of the most difficult tasks and, in the opinion of the author, the most difficult, especially for someone that is not familiar enough with computer imaging or PyTorch fundamentals. For humans, perceiving the data from the dataset is easily done today: one can just open files in the dataset with a double-click.

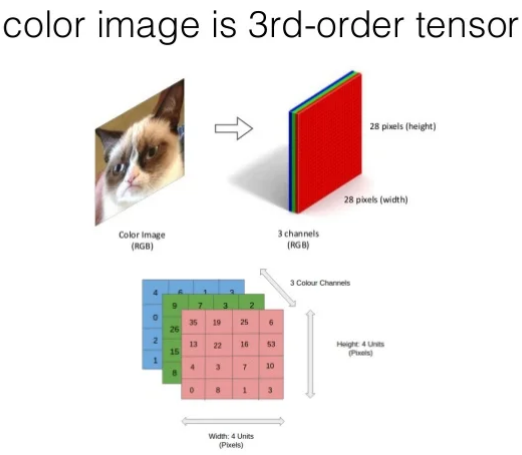

Figure 15: Visual explanation of how color images are stored in a computing system[1]

Machines, however, only perceive numbers, and so we must find a way to represent the frames from a video into numbers. This is easy, since this is how videos are actually stored anyway. To understand this, consider a grayscale, or black-and-white image: the shades of black in the image can be imagined as numbers on a scale, say from 0 to 255, the number representing the intensity of light. For a white area, where light is more intense, one would use values closer to 255, and for a black area, where light intensity is low, values closer to 0 would be used. This way, the entire black-and-white image can be imagined as matrix of values between 0 to 255.

[1] © https://lisaong.github.io/mldds-courseware/

Going further, one can imagine an RGB image as a tensor, a three-dimensional generalization of a matrix. This way, the additional direction, or depth, represents the color channel for red, green and blue respectively. This way, any point or pixel of color in the image can be represented as the unique combination of red, green and blue light intensities between 0 and 255, as shown in figure 15.

This understanding leads to the simple definition of a video as a four-dimensional tensor, where the fourth dimension represents the index of the current frame. For example, the videos in the GRID dataset are of shape `75 x 3 x 360 x 288`, where the first dimension represents the frame index, second dimension is the color channel, and the third and fourth dimensions are the width and height of the frame, respectively.

4.2.1 DataPreprocessor

With this in mind, once the program has read the video, it should:

- **turn them to grayscale**, because it makes processing simple while preserving the features of the image,
- **crop the mouth region**, so that the program can only focus on the area of the video that matters, namely the lips, and
- **normalize the video data** to obtain a pixel mean of 0 and standard deviation of 1, in order for the neural network to be able to uniformly generalize regardless of the video's brightness and initial coloring.

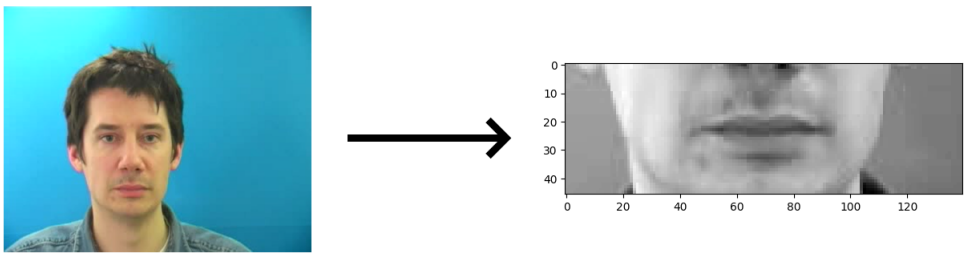

Figure 16: Visual showcase of how video data is being preprocessed, exemplified on a frame

This makes sure that the video quality is ignored and the model can focus on the relevant part of the data to learn patterns, namely the moving of the lips.

This results in video data being transformed to a tensor of shape `75 x 46 x 140`, where `75` represents the number of frames and `46 x 140` is the size of a normalized, black-and-white image containing just the mouth of the speaker. This can be better understood by observing figure 16.

Another needed functionality is the ability to load the video data and the alignment data simultaneously, and this is easily done as the two files corresponding to one data piece are named the same, except for the file type.

Similarly, the `.align` files have to be preprocessed so as to allow sentence-level lip reading. As the alignments were initially created in 2006 to allow for word-level process, which can be seen from listing 3.1, the job becomes to merge the words from the file together and ignore the timestamps and `sil` markings, and translate the sentence using a character-to-index converter function, in such a way that every letter has a corresponding number and the sentence becomes a list of numbers that can be taken by the model as the *target*.

The data is then being loaded into into the PyTorch model in a standard manner by utilizing several PyTorch-specific classes, such as `torch.utils.data.Dataset` and `DataLoader`. This data pipeline is further explained by studying the class architecture of the program at this step, which is showcased in figure 17.

The purpose of the **DataPreprocessor** class is to preprocess the data so that the model can focus on the most significant parts of the input during training.

As observed in diagram 17, *DataPreprocessor* contains the following methods:

- `__init__(mouth_crop, char_converter)`, which initializes an instance of the *DataPreprocessor* class,
- `load_mpg_file(mpg_file_path)`, which loads the data from an MPG video file,
- `load_align_file(align_file_path)`, which loads the data from a `.align` file, and
- `load_data(mpg_file_path)`, which loads the data from an MPG video file and its corresponding `.align` file by calling the above mentioned methods.

The implementation of the *DataPreprocessor* class is inspired by Renotte's LipNet implementation and adapted to use PyTorch-specific tensors as opposed to TensorFlow ones.

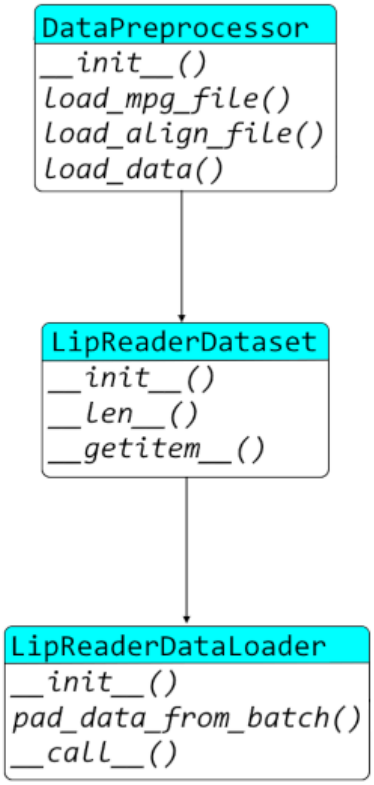

Figure 17: Diagram of *DataPreprocessor*, *LipReaderDataset* and *LipReaderDataLoader* classes, taken from figure 13

4.2.2 LipReaderDataset

The `load_data()` method is directly called by class **LipReaderDataset**, which is a concrete implementation of the dataset interface provided by PyTorch, an it overrides `Dataset`'s following methods:

- `__init__(mpg_file_paths)`, which initializes an instance of the *LipReaderDataset* by saving a list of all the MPG video files that could be opened and preprocessed using the *DataPreprocessor* class,
- `__len__()`, which returns the length of the videos in the dataset, and
- `__get_item__(index)`, which returns the preprocessed frames and alignments from the `index`'th element in the dataset by running *DataPreprocessor*'s `load_data()` method.

Lastly, this class is further utilized by **LipReaderDataLoader**, which batches and uniformly pads the data and returns the training, validation and testing dataloaders when called.

4.2.3 LipReaderDataLoader

LipReaderDataLoader implements the following methods:

- `__init__(mpg_dir_path)`, which initializes an instance of the *LipReaderDataLoader* by saving a list of all the MPG video files that are present in the `mpg_dir_path` directory,
- `pad_data_from_batch(batch)` which, given a batch, pads the data at the size of the largest element from the batch and it therefore makes sure that all data will be of the same size, and
- `__call__()`, which creates a dataset containing all the files in `mpg_dir_path`, splits it into training and validation and creates PyTorch *DataLoaders* that batch and pad the data.

This sophisticated data pipeline ensures that data is preprocessed to make the most use of what is being served as data, while generalizing the process well enough to easily allow for an increase of the size of the dataset being provided.

4.3 Create and train model

For a machine learning based solution, this is by far the most important and the most interesting part of the process. While preprocessing and decoding data can be rewarding, it is the opinion of the author that nothing compares to engineering a study neural network architecture.

However, the current section, which mainly concerns the whole system's architecture, will not focus on the design of the neural network and will rather treat it as a *black box*.

The details of the LipLink neural network architecture, as well as an explanation for the layers and hyperparameters used, will be tackled in depth at the end of this section.

4.3.1 LipReader

The model architecture is stored inside the **LipReader** class, which implements the PyTorch-specific `torch.nn.Module`. As it can be observed in figure 18, it only contains two methods:

- `__init__(num_classes)`, which initializes an instance of the *LipReader* by defining the layers that will be used by the model, and

- `forward(x)` which, given an input batch, passes it through the model.

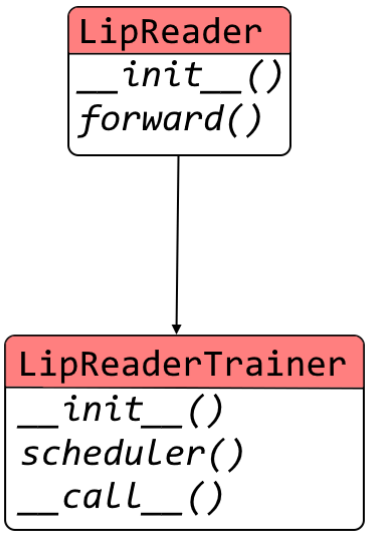

Figure 18: Diagram of *LipReader* and *LipReaderTrainer* classes, taken from figure 13

This sort of arrangement is the *canonical* way of defining a neural network model in PyTorch.

4.3.2 LipReaderTrainer

The model is then instantiated by **LipReaderTrainer**, which effectively trains the model to interpret the movement of the lips.

The training effectively means that *LipReaderTrainer* passes the preprocessed data batches through the randomly-initialized model to obtain an output, then computes how well the model performs using a *loss* function and then uses this result to update the model's parameters so that they are able to generalize the input and perform better the next time.

This process is called an *epoch*, and to train a model with a good enough accuracy, especially for such a complex pattern matching and when the dataset is not very large, as we will see, one generally needs a multitude of epochs.

Class *LipReaderTrainer* is the most complex one yet, although the architecture has been minimally-built as in to better avoid mistakes, and contains the following methods:

- `__init__(train_data_loader, test_data_loader, model, learning_rate, checkpoint_dir_path, epochs, char_converter)`, which initializes an instance of the *LipReaderTrainer* by defining the data loaders, model and hyperparameters that will be used,

- `scheduler(optimizer, epoch, initial_learning_rate, decay_rate)` which updates the learning rate as training progress, so as to make sure that the model trains to its maximum potential and does *overfit* i.e. learn the inputs by heart, and
- `__call__()`, which simply trains LipLink for `epochs` number of epochs and with the wanted parameters.

LipReader and *LipReaderTrainer* are where the magic happens, and this is why their detailed architecture and processes will be discussed in detail at the end of this section.

4.4 Decode data

All that training the model does can be for nothing if the predictions it outputs cannot be understood by humans. This is where the step to decode the data comes in.

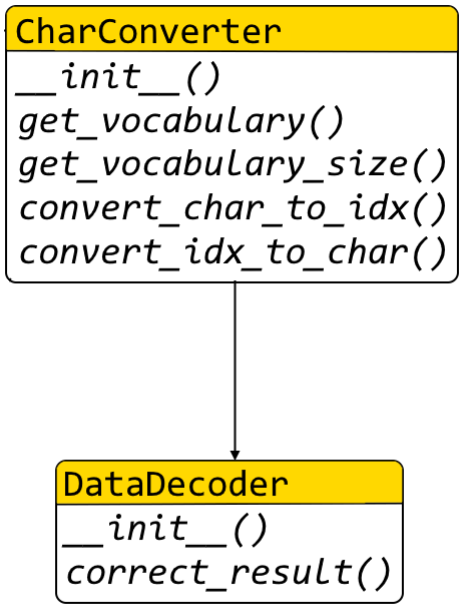

Figure 19: Diagram of *CharConverter* and *DataDecoder* classes, taken from figure 13

4.4.1 CharConverter

The first and most important decoder class is **CharConverter**. *CharConverter* is the actual encoder-decoder class used by many processes throughout the program to translate the targets, in our case the alignments from the `.align` files, from characters to indices and vice versa.

CharConverter is modelled based on `tensorflow.keras.layers.StringLookup`, which is a TensorFlow-specific implementation of converting chars to numbers from a vocabulary given as argument. Since PyTorch does not have an equivalent to it, *CharConverter* had to be implemented from scratch.

As seen in figure 19, *CharConverter* implements the following methods:

- `__init__(vocabulary, oov_token)`, which initializes an instance of the *CharConverter* class,
- `get_vocabulary()`, which returns the vocabulary of the converter as a string of characters,
- `get_vocabulary_size()`, which returns the size of characters in the vocabulary, plus the out-of-vocabulary token,
- `convert_char_to_idx(char)`, which converts a string stored in different ways to the list of its corresponding indices, and
- `convert_idx_to_char(idx)`, which does the opposite conversion.

The *vocabulary* is defined as a continuous string of the characters that can appear in the sentences that are being said in the dataset, and in our case contains the entire English-language characters, a few punctuation marks, digits from 0 to 9 and space, for a total of 39 characters.

To it, we add the *out-of-vocabulary token*, which will be the token that the model will predict for the frames in which it does not detect a sound. For simplifying the output, we use the empty string, `""`, as the out-of-vocabulary token.

Taking the out-of-vocabulary character into consideration as well, we obtain a total of 40 characters that can be predicted for any frame. This is why the problem of automated lip reading, at least in the context we attempt to solve it, can be thought of as a classification problem with 40 classes from which the model has to predict for every frame in a video.

4.4.2 DataDecoder

This realization leads to a problem that can appear during the testing phase. Namely, as shown by the saliency maps depicted in figure 9 provided by Assael et al. [1], characters that encompass a word can be predicted more than once, due to there being more frames one after the other for which the prediction is the same character. For example, the word `please` can be predicted as `pleeaasse`, which a human can definitely interpret as the initial word, but it can definetely impact our accuracy and increase our WER.

This is why the **DataDecoder** class is needed, as its main purpose is to correct the *typos* that can plague the model's predictions. For this, it uses the `textblob` Python package,

which contains a convenient method that when called on an English-language string returns the string without the typos that it may or may not contain.

The *DataDecoder* class implements the following methods:

- `__init__()`, which initializes an instance of the *DataDecoder* class, and
- `correct_data(data)`, which returns the string given in `data` without its corresponding typos.

4.5 Test model

With everything in place and the model trained, all we need to do is to test it to observe how it behaves. Testing is the final part of our five step process, and it is the one that can result in further modifications in the approach used during the previous steps.

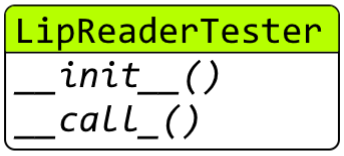

Figure 20: Diagram of *LipReaderTester* class, taken from figure 13

4.5.1 LipReaderTester

As it can be observed from figure 20, **LipReaderTester** contains only two methods:

- `__init__(test_data_loader)`, which initializes an instance of the *LipReaderTester* class, and
- `__call__()`, which computes different statistics to see how the model behaves on the dataset.

The most relevant to us are the *word-level accuracy* or opposite to WER (word error rate), defined as the number of words from the testing dataset that were predicted correctly, and the *edit distance*, to determine how many characters were wrongly predicted in a given sentence.

LipReaderTester is the last concrete class that completes the five step approach described in the beginning of this chapter and allows us to go on to discuss the neural network model architecture and hyperparameter configuration.

4.6 Model Architecture

The LipLink model, defined in the *LipReader* class that was described above and based on the automated lip reader architecture described by Yannis M. Assael et al. [1] in their 2016 paper, is, in the opinion of this paper's author, the crowning achievement of months of research in the field of machine learning, and this is why it deserves to be covered more in depth in the following sections.

Similar to the structure of LipNet depicted in figure 8, LipLink's model architecture is complex, featuring three phases: (1) convolutional, (2) recurrent and (3) fully-connected.

The *model's architecture* is depicted in-depth by figure 21.

The **convolutional phase** consists of three spatio-temporal convolutional layers, each convolutional layer being followed by a rectifier (ReLU) activation and a `2 x 2` spatial pooling. Intuitively, the convolutions play the role of filtering visual patterns and their appearence in the data. The ReLU activation introduces non-linearity to the data, while the pooling operations have the goal of minimizing the samples' size of these layers' output.

Afterwards, the **recurrent phase** of the network starts by flattening the data into a one-dimensional tensor that is suited for GRU. GRU, or Gated Recurrent Unit, is a recurrent neural network layer architecture that stores the long-term temporal dependencies of the different movements in the video and gives the neural network its ability to *contextualize* its predictions. After every GRU layer, a dropout is applied to prevent overfitting.

Lastly, the **fully-connected phase** of the model is a simple fully-connected layer that maps the combined interpretation of the previous phases into one of the possible classifications for the sound that is being said in the current frame.

At the end, the loss function that is being used to update the gradients is the **Connectionist temporal classification** (CTC). This loss function is widely used in speech recognition today because, according to LipNet's authors [1], it eliminates the need for aligning the input data to the target when the input's timing is variable. The optimizer used to update the gradients is **Adam** with an initial learning rate of `0.0001` that decays exponentially after epoch 30.

This complex architecture, containing around 1.4 million parameters, is able to generalize the data provided from the GRID dataset without overfitting and achieves an accuracy of 88.04% on the testing dataset. The subject of the following sections becomes to further understand how every method that was used during the creation of this architecture functions.

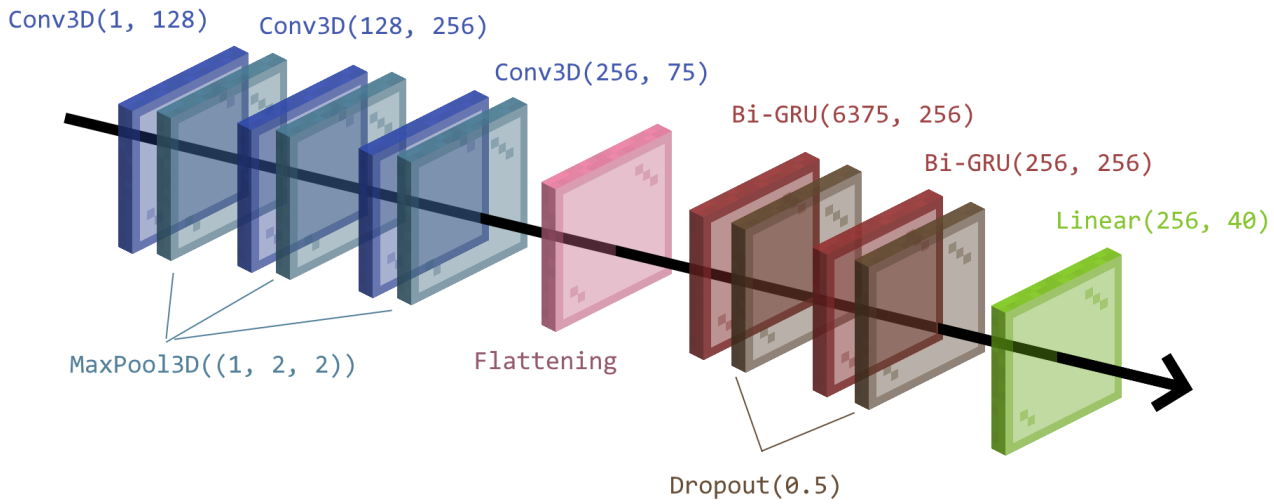

Figure 21: LipLink *model architecture diagram*, showcasing every layer used in the design of the model, as well as its most important parameters[2]

[2]© The layer square colored images were taken from https://minecraft.wiki/w/Stained_Glass_Pane

4.7 CNN

4.7.1 Convolution

According to Ian Goodfellow's seminal book on the subject of deep learning [7], a convolution is, broadly, the mathematical operation that combines two functions to create a third.

$$s(t) = \int x(a)w(t-a)\,da \tag{1}$$

For us, who work with concrete data typically stored in matrices, such is the case of images, the convolutional operation typically involves centering a smaller matrix called a *kernel* onto the larger image we are applying the operation on and repeatedly calculating the multiplication between the kernel and a smaller area of the matrix and adding up the elements of the resulting matrix to obtain a value in the resulting matrix.

Mathematically, this looks something like:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{2}$$

For a better visual explanation, one could check figure 22 that depicts how it is done on an example.

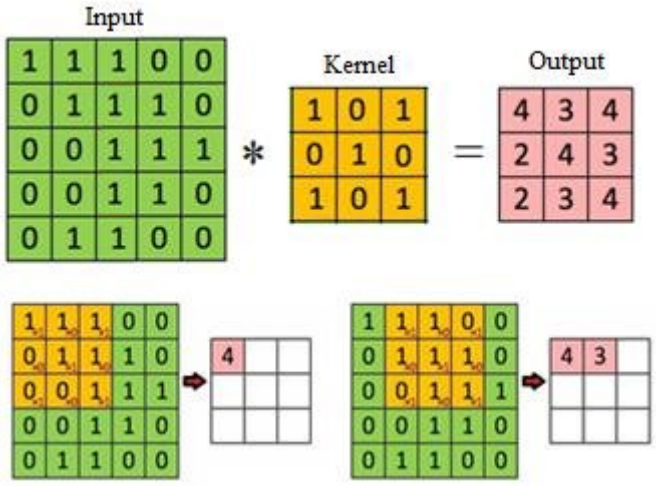

Figure 22: Visual representation of a convolution[3]

The reason why the convolution operation, although simple, has been at the forefront of deep learning in the last few years is that the application of different kernels on an image matrix produces data that is quite literally enhancing a certain feature from the original image.

Under today's deep learning framework of understanding, convolutions are how a computer sees. The achievement however is no small feat and training a model to detect complex patterns in an image require a large amount of filters, and the multiplications that they need lead to vast needs of computational power that are today solved by GPUs.

According to Ian Goodfellow [7], the convolutional layers of today's deep neural networks are the closest way humans have been able to mimic the neural structure of our brain in modern engineering. The neurons in the human visual cortex are of different types, some used to detect simpler features (e.g. lines, squares, circles) while others building on top of the first ones to understand more complex features (e.g. faces, snakes, cars). Convolutional neural networks work in a similar manner, with the first layers detecting simple features while the last layers usually identifying complex features from the image, such as digits or animals.

Building on this understanding, a convolutional layer has certain parameters that it depends on, and we can name:

- **kernel size**, representing how large the kernel that gets applied to input data is,
- **padding**, representing whether the initial image is padded with zeroes around it so that the convolution operation can be centered on the margin elements as well,
- **stride**, representing the distance the kernel moves during the operation, with the default usually being 1, and
- **dilation**, or the spacing between kernel matrix elements. Using dilation, the kernel gets larger but the number of parameters stays the same.

The size of the output of a convolutional layer can be calculated using the following formulas:

$$O_H = \left\lfloor \frac{H + 2P_H - K_H}{S_H} \right\rfloor + 1 \tag{3}$$

$$O_W = \left\lfloor \frac{W + 2P_W - K_W}{S_W} \right\rfloor + 1 \tag{4}$$

where:

- H and W are the height and width of the input,
- K_H and K_W are the height and width of the kernel,
- P_H and P_W are the padding applied to height and width and
- S_H and S_W are the stride along the height and width, respectively.

[3]© https://www.researchgate.net/figure/Convolution-Operation-on-a-5x5-Matrix-with-a-3x3-Kernel-Zero-Padding-Convolution-layer

For most tasks that pertain to a machine learning solution today, convolutional neural networks (CNNs) are usually the central piece of the provided solution, and considering their already discussed advantages, it is easy to see why.

4.7.2 STCNN

The spatio-temporal convolution that constitutes spatio-temporal convolutional neural networks (STCNNs), also known as a *three-dimensional convolution*, is a generalization of the convolution operation for 3D tensors, such as videos. It is utilized when we want the filtering to also take into account the change between frames over time, such is the case of what we want to achieve for lip reading.

According to Yannis M. Assael [1], we can formally define the ST convolution operation with the following formula:

$$[stconv(\mathbf{x}, \mathbf{w})]_{c,t,i,j} = \sum_{c=1}^{C} \sum_{t=1}^{k_t} \sum_{i=1}^{k_i} \sum_{j=1}^{k_j} w_{c,t,i,j} x_{c,t+t',i+i',j+j'} \tag{5}$$

The STCNN performs the convolution operation using a 3D tensor and filters both spatial and temporal patterns that are needed for lip reading.

4.7.3 ReLU

The convolution operation is a linear operation, and a simple convolutional layer normally allows the network to model a linear relation between the inputs and the target outputs. However, it is the case of many real-life relations between different types of data that they are non-linear, and by using just convolutional layers alone, one could not recreate them.

This is when activation functions come in, and their goal is to solve this problem. By adding a ReLU activation after every convolution, which is what we do in LipLink, we aim to introduce non-linearity in our model, so that the input data can be better correlated to the desired outputs.

4.7.4 Pooling

Pooling is usually discussed [7] as an operation whose goal is to lower the complexity of an input by downsampling it so that the output of such a layer is of smaller size than the input,

but the usual shape of the filter is preserved.

This is usually done by taking contingent parts of the input tensor and applying a pooling function to them. There are many pooling functions that can be used, but by far the most common ones are *max-pooling*, when the maximum value in the selected region of the input is chosen as the output of that region, and *average pooling*, when the average of values in the selected region is chosen.

A note that the uninitiated reader must pay attention to is that ReLU and pooling are not neural network layers in the traditional sense of the word, because they do *not* have trainable parameters. While convolutional layers have weights and biases that, through training, we look to optimize so that they can produce the targets through inputs alone, ReLU and other activation functions, as well as max-pooling or average pooling, are simple operations that get applied in the same manner on their input every time.

4.7.5 Convolutional phase

During the convolutional phase of the model, which is detailed visually in 23, we utilize what we learned about 3D convolution, ReLU and max-pooling in order to create a neural network that can extract the complex features of moving lips.

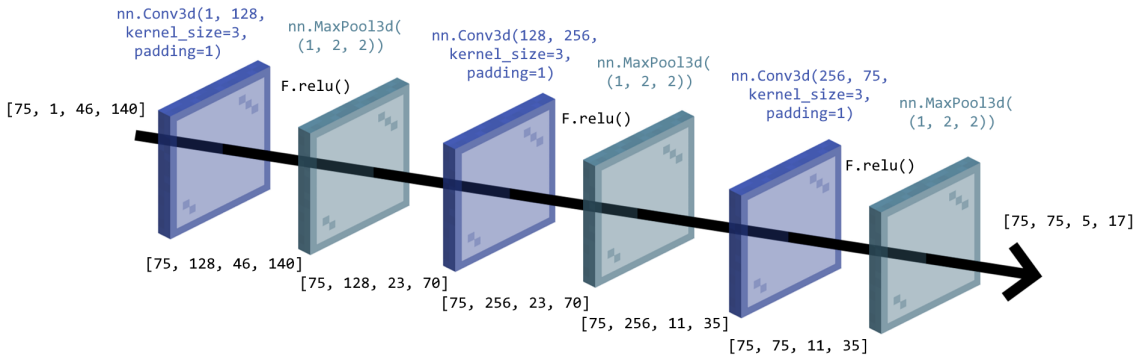

Figure 23: Diagram of the *convolutional phase* of the LipLink model, consisting of 3 convolutional, ReLU activation and max-pooling layers

For the input tensor of size `75 x 1 x 46 x 140`, where 75 is the number of frames, 1 is the channel dimension for the grayscale image, and `46 x 140` is the width-height size of a frame from the video, one passes through the convolutional phase of the network by going through the following layers:

- `torch.nn.Conv3d(1, 128, kernel_size=3, padding=1)`, the first spatio-temporal convolutional layer that outputs a tensor of size `75 x 128 x 46 x 140`,

- `torch.nn.functional.relu()`, the first application of the ReLU activation function that does not change the size of the output,
- `torch.nn.MaxPool3d((1, 2, 2))`, the first 3D max-pooling layer applying a `1 x 2 x 2` pooling kernel on the spatial dimension of its input and effectively cutting its size in half,
- `torch.nn.Conv3d(128, 256, kernel_size=3, padding=1)`, the second spatio-temporal convolutional layer
- `torch.nn.functional.relu()`, the second application of the ReLU activation function,
- `torch.nn.MaxPool3d((1, 2, 2))`, the second 3D max-pooling layer that cuts the width and height of the frames in half
- `torch.nn.Conv3d(256, 75, kernel_size=3, padding=1)`, the third spatio-temporal convolutional layer
- `torch.nn.functional.relu()`, the third application of the ReLU activation function, and
- `torch.nn.MaxPool3d((1, 2, 2))`, the third 3D max-pooling layer whose applying results in a final tensor of size `75 x 75 x 5 x 17`.

The input and output of every layer in the convolutional phase of the model is indicated by table 2.

Table 2: Output size of every layer of the convolutional phase of LipLink

Convolutional layer	**Output size**
`torch.nn.Conv3d(1, 128, kernel_size=3, padding=1)`	`75 x 128 x 46 x 140`
`torch.nn.functional.relu()`	`75 x 128 x 46 x 140`
`torch.nn.MaxPool3d((1, 2, 2))`	`75 x 128 x 23 x 70`
`torch.nn.Conv3d(128, 256, kernel_size=3, padding=1)`	`75 x 256 x 23 x 70`
`torch.nn.functional.relu()`	`75 x 256 x 23 x 70`
`torch.nn.MaxPool3d((1, 2, 2))`	`75 x 256 x 11 x 35`
`torch.nn.Conv3d(256, 75, kernel_size=3, padding=1)`	`75 x 75 x 11 x 35`
`torch.nn.functional.relu()`	`75 x 75 x 11 x 35`
`torch.nn.MaxPool3d((1, 2, 2))`	`75 x 75 x 5 x 17`

4.8 RNN

After having identified the spatio-temporal features of the videos from the training dataset, the model needs to be able to deduce long-term relations, or contexts, that occur between

these features in order to be able to predict what is being said at the sentence level.

RNNs are all about that: when predicting a time series, instead of just relaying on their current inputs, RNNs also have a hidden state that they use to keep track of their previous output. This way, they can use what they interpreted about the former elements of a time series in their interpretation of the current element in the time series.

Recurrent neural networks are named like that because they are cyclical graphs (have loops). Contrary to the feed-forward fashion of information transmission inside a CNN, RNNs previous predictions in a time series influence their current prediction.

However, according to Goodfellow [7], deploying vanilla RNNs has one problem that is commonly called the *vanishing gradient problem*. During training, when computing the gradients of the loss function so as to backpropagate and update the weights accordingly, the mathematical nature of these gradients makes it that, after a while, they are explode (grow exponentially) or vanish (shrink exponentially). This can literally halt the process of training, no matter how many epochs pass.

4.8.1 GRU

This is one of the many reasons why GRUs, or Gated Recurrent Units, were developed. As seen in their name, their new addition to simple RNNs is their inclusion of *gates*. These gates regulate what information is being passed through and out of the network and helps it remember long-term patterns more clearly.

The standard formulation of the GRU has two gates:

- **Reset gate**, which decides how much of the previous information to lose
- **Update gate**, which decides how much of the previous information to change
- **New memory content**, which combines the reset gate with the new input so that it loses some of the new information
- **Final hidden state**, which combines the update gate with the new memory content so that it constitutes the final hidden state of the unit

Assael et al. [1] define the standard mathematical formulation of the GRU to be:

$$[u_t \; r_t]^T = \sigma \left(W_z z_t + W_h h_{t-1} + b_g \right) \tag{6}$$

$$\hat{h}_t = \tanh \left(U_z z_t + U_h (r_t \odot h_{t-1}) + b_h \right) \tag{7}$$

$$h_t = (1 - u_t) \odot h_{t-1} + u_t \odot \hat{h}_t \tag{8}$$

The Gated Recurrent Unit is complex, and only appeared in 2014. [1] Before then, the preferred RNN similar to GRU was LSTM (Long Short Term Memory), which has been the prevailing RNN technology during the 2000s. The LSTM has three gates instead of two and is therefore harder to train. Due to its simplicity and proven use cases, the GRU has become the go-to for RNNs, and is what we are going to use.

4.8.2 Bi-GRU

The Bidirectional GRU, or Bi-GRU is an implementation of the GRU that, simply put, takes the hidden state information not only from the previous time series inputs, or the past context, but also from the *future* context i.e. the following time series inputs.

It does this by employing two GRUs: one to process inputs in a forward direction, and one to process them in a backward direction.

This can be extremely useful when dealing with data that could utilize both the past and future contexts it finds itself in, such as speech. This is why Bi-GRU are used today for many speech recognition tasks, and we will be using it for LipLink as well.

4.8.3 Flattening

However, an important thing to note is that, before we can feed the data from the convolutional phase into our GRUs, it needs to be flattened. For every element (frame) in the time-series (video), the GRU expects a continuous, one-dimensional array of features as its input.

From the output of the convolutional phase of `75 x 75 x 5 x 17`, the input to the Bi-GRU layers has to be of shape `75 x 6375`. Basically, we are storing the convolutional output of every frame as an array. The GRU takes 75 such arrays as its time-series, one for every frame in the input video.

4.8.4 Dropout

Another problem with GRUs, especially in our context of using a relatively small dataset, one common problem is that they can learn this data so well that the model fails to generalize anymore, and performs poorly on new, unseen data. This is called *overfitting*.

One technique used to limit overfitting is the introduction of Dropout layers in the model's

architecture. What this does is that it randomly zeroes a section of the neurons. This basically means that, at every forward pass through the network, a certain amount of neurons will be set to 0. The remaining neurons are scaled so that the output is usually not affected by the dropping.

To prevent overfitting, we will include a Dropout layer that drops 50% of the neurons after every GRU.

4.8.5 Recurrent phase

During the recurrent phase of the model, which is detailed visually in figure 24, we first flatten the data and then utilize 2 Bi-GRU layers, each followed by a 50% dropout layer. This setup is good enough for the model to learn the possible lip movements from the dataset and what each one of them means.

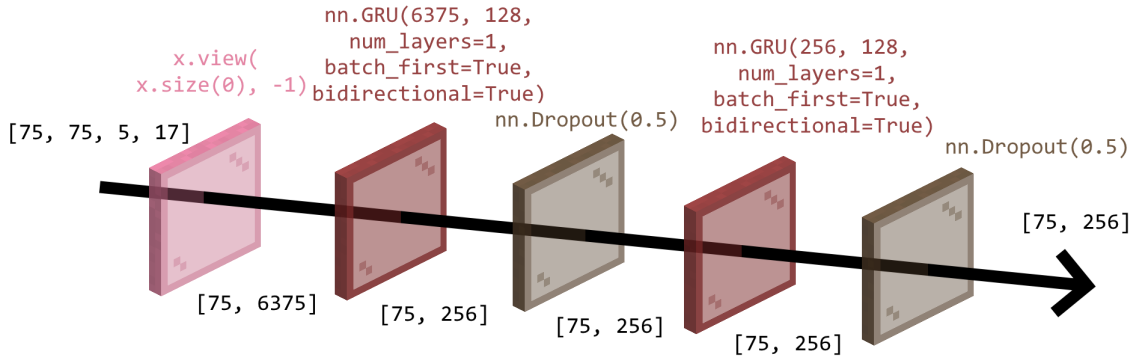

Figure 24: Diagram of the *recurrent phase* of the LipLink model, consisting of a flattening followed by two Bi-GRU and dropout layers

For the convolutional phase output of size `75 x 75 x 5 x 17`, where 75 is the number of frames, 75 is the number of features from a frame, and `5 x 17` is the width-height size of a feature, one passes through the recurrent phase of the network by going through the following layers:

- `x.view(x.size(0), -1)`, the one-dimensional flattening of the features that turns the tensor into `75 x 6375`,
- `torch.nn.GRU(6375, 128, num_layers=1, bidirectional=True)`, which is the first bidirectional GRU layer,
- `torch.nn.Dropout(0.5)`, the first dropout layer,
- `torch.nn.GRU(256, 128, num_layers=1, bidirectional=True)`, the second Bi-GRU, and

- `torch.nn.Dropout(0.5)`, the second dropout layer, whose applying leads to a final tensor of size `75 x 256`.

The input and output of every layer in the recurrent phase of the model is indicated by table 3.

Table 3: Output size of every layer of the recurrent phase of LipLink

Recurrent layer	**Output size**
`x.view(x.size(0), -1)`	`75 x 6375`
`torch.nn.GRU(6375, 128, num_layers=1, bidirectional=True)`	`75 x 256`
`torch.nn.Dropout(0.5)`	`75 x 256`
`torch.nn.GRU(256, 128, num_layers=1, bidirectional=True)`	`75 x 256`
`torch.nn.Dropout(0.5)`	`75 x 256`

4.9 Linear

The results of the recurrent phase of LipLink is a list of 256 numbers for every submitted frame. These 256 numbers indicate what character from the vocabulary can be associated with that frame from the video. To make this last association, we need a linear correlation between them and the possible classes, in our case characters from the vocabulary.

This is where we use a fully-connected layer.

4.9.1 Fully-connected phase

As can be observed visually in figure 25, the last section of our neural network uses one single fully-connected layer to correlate the patterns from the previous phases to one of the possible characters from the vocabulary.

All the neurons from the fully-connected layer are connected to all the neurons in the previous layer, as one could guess from its name.

To predict more complex patterns, Ian Goodfellow, in his book [7], argues for the use of two fully-connected layers at the end of a neural network, however, due to the fact that our patterns should be easy enough to correlate, our implementation uses only one such layer.

For the recurrent phase output of size `75 x 256`, one passes through the fully-connected layer by going through a layer `torch.nn.Linear(256, 40)`, and as such the final output

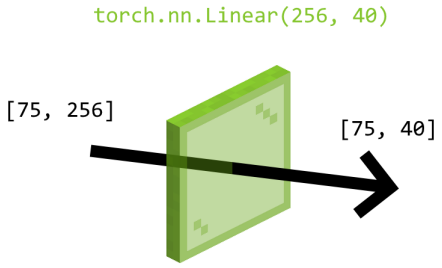

Figure 25: Diagram of the *fully-connected phase* of the LipLink model, consisting of a single fully-connected (linear) layer

has the size `75 x 40`. For every of the 40 classes, the model predicts the probability that the frame corresponds to each of these classes. Applying the softmax activation of these results with `torch.log_softmax()` and selecting the class with the largest probability with `torch.argmax()` will lead us to the most probable corresponding character for the frame.

4.9.2 Softmax

The need for *softmax* comes from the following problem: the output of the fully-connected layer is in logits. We are interested in converting these logits to probabilities, so that we can interpret the prediction of the model accordingly.

This is what softmax does. For an array `z` of `K` logits, we calculate the softmax for every element in the array as:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{9}$$

This effectively returns the probability distribution over `K` classes, which is what we were looking for.

4.10 CTC loss

Using CTC loss, short of *Connectionist temporal classification*, is one of the most important pieces of the puzzle that is our model's architecture. What makes CTC loss relevant is that it allows for measuring the rate of error between an output and a target when the data that they contain is not aligned.

This makes it incredibly well-suited for tasks where the result is not aligned and can come at different positions in the prediction, but what is important is that it comes. This is the case of most speech recognition systems, because, intuitively, it does not matter which sequences predict what words, what matter is only that they do.

For us trying to achieve automated sentence-level lip reading, this means that we can consider the cases where adjacent frames produce the same output and the cases where the expected word is predicted a few frames left or right as valid.

4.11 Adam optimizer

Similar to Renotte's LipNet, LipLink's implementation uses the Adam optimizer with an initial learning rate of `0.0001`. This decreases exponentially after epoch 30 so as to make sure the large learning speed does not lead to overfitting.

The value of the learning rate for the first 100 epochs can be observed in the graph from figure 26.

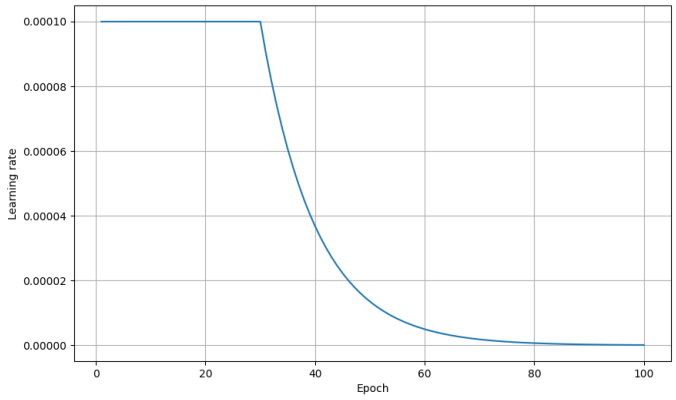

Figure 26: The learning rate used by the Adam optimizer for the first 100 epochs

5 DESIGN AND IMPLEMENTATION

This section builds upon the previous by following the implementation of LipLink closely and detailing the difficulties that appeared during the development of the above-mentioned solution, as well as the steps that were taken to mitigate them.

Machine learning is a field that is notorious for it being hard to produce good results in a narrow time frame, with debugging and finding the cause of the problems being hard to do, especially for beginners. Considering its prior experience with delivering software that does not use machine learning, the author noted that the development process and detecting the problems was incredulously hard, and at times even frustrating.

The problems that were encountered are described in the following sections, in the order that they appeared based on a linear implementation of the solution described above, and there is also a section concerning the experimenting that was done on the neural network model, and its associated results.

5.1 Data not fetching correctly

The first problem encountered was that, when downloading the GRID dataset as described in the *Fetch data* section, the download would stop randomly and the resulting file would be corrupted. This situation was extremely unwelcome considering the fact that the dataset is a rather large data file, containing 12.9 GB, and that the goal was to build an automated data pipeline that the users could utilize to download it.

The problem was pinpointed as being caused by the Internet connection between the server hosting the data and the personal client machine that was trying to keep being connected to it for the entire time of downloading.

This is why the solution that was proposed was what the author termed *download-in-the-loop*, which consists of downloading, checking whether the downloaded file was complete and not corrupted and reattempting in the case it is not. This process is exemplified in listing 5.1.

```python
while True:
    # Download the file
    response = requests.get(self.data_url, stream=True)
    response.raise_for_status()

    # Write the file to the download path in chunks
    with open(self.data_file_path, "wb") as data_file:
        for chunk in response.iter_content(chunk_size=8192):
            data_file.write(chunk)

    # Check if the downloaded file matches the expected size
    if os.path.getsize(self.data_file_path) == self.data_file_size:
        break
```

Listing 5.1: Example of how download-in-the-loop could be implemented using Python

This will eventually lead to a correct download, in the experience of the author, after a couple of attempts.

5.2 Data normalization

As discussed in chapter 4, in order for the model to be able to better generalize data, such as video data, it is good practice to normalize it to a mean of 0 and standard deviation of 1. This can be swiftly achieved by utilizing the following formula:

$$data[x, y] = \frac{data[x, y] - mean}{std} \tag{10}$$

Nicholas Renotte's LipNet [9], which is the main inspiration for the *DataPreprocessor* class, attempts to do this as well using TensorFlow tensors but his implementation makes a classical mistake of integer to floating-point conversion, which can be seen in listing 5.2.

```
mean = tf.math.reduce_mean(frames)
std = tf.math.reduce_std(tf.cast(frames, tf.float32))
return tf.cast((frames - mean), tf.float32) / std
```

Listing 5.2: Renotte's LipNet's wrong data normalization technique based on TensorFlow

The issue lies with the casting of frames to `tf.float32`, that is done for computing the standard deviation but is not saved. So, before the `return` line, variable `frames` is still of type unsigned integer, and substituting the mean from the numbers below the mean effectively leads to values close to 255 instead of negative values, adding white noise to the videos.

The correct way to do it, which is how it is done in LipLink using PyTorch, is casting it in-place, as in listing 5.3.

```
frames = torch.tensor(frames, dtype=torch.float32)
mean = frames.mean()
standard_deviation = frames.std()
frames = (frames - mean) / standard_deviation

return frames
```

Listing 5.3: LipLink normalization technique based on PyTorch, making sure the casting to float is saved

5.3 Building a data pipeline

A data pipeline is needed so that one can easily feed data into the model for training and testing. After implementing the preprocessing methods for both inputs and targets, I attempted to build the pipeline similar to the TensorFlow one created by Renotte [9], only utilizing objects that I was most familiar with, such as Python lists and `torch.Tensor` objects.

As it turns out, this approach is not only inefficient, but incomplete, and keeping large data quantities in such a way will only backfire. This was a key moment when I turned to reseaching PyTorch, and I realized that the standard way of using `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` was much more effective.

The clean way to do it is to implement a concrete dataset that inherits `Dataset` and implements its `__get_item__()` method by returning a preprocessed tuple of an input and a target. This Dataset, which can be split into training and validation as we will see later, is then used by `DataLoader` to create a DataLoader object that can be used during training and testing, as it contains all data in the dataset.

The data is loaded and preprocessed at runtime when a new tuple from the DataLoader is needed, as can be seen in listing 5.4, and not beforehand, greatly removing large memory needs and delays that were caused by pre-loading the data.

```
class LipReaderDataset(Dataset):
    def __init__(self, mpg_file_paths, data_preprocessor):
        self.mpg_file_paths = mpg_file_paths
        self.data_preprocessor = data_preprocessor

    def __getitem__(self, index: int):
        # Get the path to the MPG file
        mpg_file_path = self.mpg_file_paths[index]

        # Load the data from the MPG and the align file
        frames, alignments = self.data_preprocessor.load_data(
            mpg_file_path)

        return frames, alignments

# Create the data set
lip_reader_dataset = LipReaderDataset(self.mpg_file_paths)

# Create the data loader
lip_reader_data_loader = DataLoader(lip_reader_dataset, batch_size
    =2, shuffle=True)
```

Listing 5.4: LipLink concrete use of PyTorch Dataset and DataLoader for creating a data pipeline

5.4 Layer dimensions

It is important to note that batching the data, as can be seen in listing 5.4, adds an additional dimension to our input data tensor, making it become `2 x 75 x 1 x 46 x 140`. The classical solution is to extract the elements from the batch before running them through the network, however, the more elegant solution is to adapt one's network to the batched inputs.

This adaptation took the author a while because they had to manually check the dimensions from one layer to the other and, at the time of engineering the model's architecture, they would change quite often and not be set in stone. However, the only change that had to be taken was modifying the flattening layer, from `x = x.view(x.size(0), -1)`, as discussed above, to `x = x.view(x.size(0), x.size(1), -1)`. This keeps the batch and frame dimensions and flattens the rest, essentially turning a `2 x 75 x 75 x 5 x 17` to a `2 x 75 x 6375`.

5.5 Splitting data into training and validation

This was a point of contention for the author for almost as much as they worked on implementing LipLink. Usually, when splitting a dataset into training and testing, something like the Pareto ratio of 80/20 is applied. Nowadays, Andrew Ng's ratio of 60/20/20 for training/validation/testing has become increasingly popular.

The problem with utilizing such a ratio was the imposition of utilizing only a small subset of the GRID dataset due to only having access to limited computational capabilities. The GRID dataset is huge and, even though the author initially wanted to use it in its entirety, training on the entire dataset could take up to a week. This is why the dataset utilized by LipLink only contains the 1000 videos produced by the first speaker, `s1`. Even training on the 1000 videos using a NVIDIA GeForce RTX 2070 takes around 9 hours to complete 100 epochs. However, this imposition of limiting the dataset aligns with the pillar of motivation mentioned in chapter 2 of creating an application that is simple and accessible to a larger audience, including those with limited computational capacity.

This makes it so that the 1000 videos from the narrowed GRID dataset must be used to their full potential of learning, and keeping only 600 for the training would greatly impact the model's learning ability. This is why the author took the decision of splitting the dataset 90/10, with 90%, or 900 videos being used exclusively for training, while the other 10% of videos being used for validation and testing purposes.

We recognize that increasing the dataset size, while acquiring even more diverse data, could increase the model's accuracy significantly, as demonstrated by Pingchuan et al. [4].

5.6 Dropout layers

Another problem that was overfitting the model was utilizing one instance of `torch.nn.Dropout` after both GRU layers, as the author initially thought that Dropout is not trainable and therefore the same instance could be used everywhere in the model, as in listing 5.5.

The reality is each Dropout layer instance chooses what units to drop randomly, but using the same layer at two different places in the network would drop the same neurons at both places, effectively making the two Bi-GRU configuration act like a one Bi-GRU.

```
class LipReader(nn.Module):
    def __init__(self, num_classes):
        [...]

        self.dropout = nn.Dropout(0.5)

        [...]

    def forward(self, x):
        [...]

        # GRU layers
        x, _ = self.gru_1(x)
        x = self.dropout(x)
        x, _ = self.gru_2(x)
        x = self.dropout(x)

        [...]
```

Listing 5.5: LipLink model wrongly utilizing only one dropout instance

5.7 Experiments

The main point of experimentation in choosing the final model revolved around the recurrent phase of the network and whether its implementation was optimal.

For this, three experiments were conducted, as follows:

- **experiment 1**, in which only 1 bidirectional GRU was used in the recurrent phase of the network
- **experiment 2**, in which 2 bidirectional GRUs were used in the recurrent phase of the network
- **experiment 3**, in which 2 bidirectional LSTMs were used in the recurrent phase of the network

What these components are and how they were has been briefly touched upon in the chapter 4 of this work.

5.7.1 Experiment 1: Using one Bi-GRU

The author initially considered that using only one bidirectional GRU would be enough to generalize the small amount of words that are being uttered in the GRID dataset, with a good accuracy.

This experiment was attempted with the idea that this simplier architecture would serve well our goal of providing a simplified approach to the LipNet architecture.

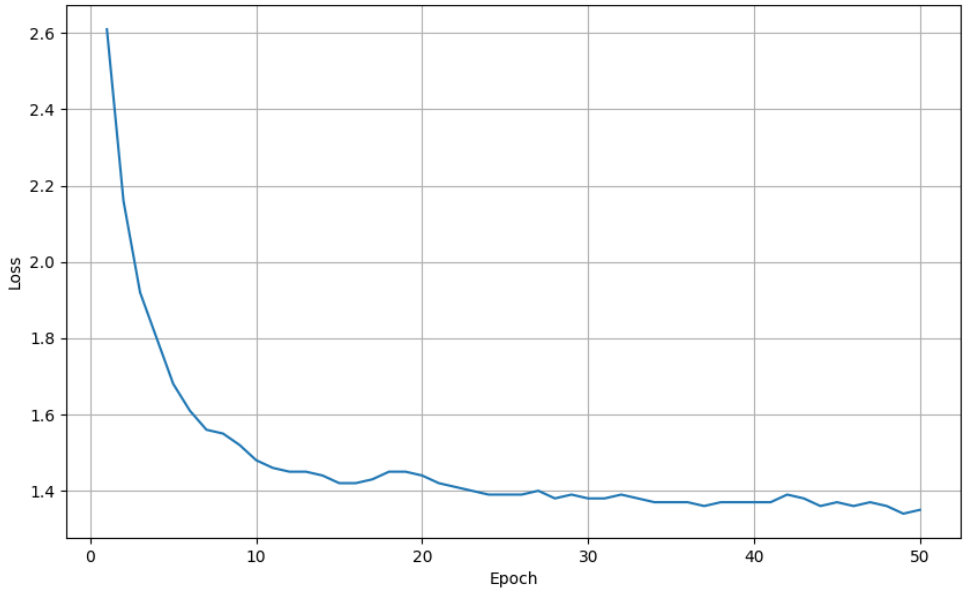

Figure 27: Training loss of experiment 1 with one Bi-GRU architecture on the first 50 epochs

However, this was not proven true. As it can be observed in figure 27, the average training loss starts at around `2.6`, which is an expected value for CTC loss with our model utilizing randomly generated weights, as it does in the beginning, and steadily declines during the first 10-15 epochs. After this, the training loss plateaus around `1.35-1.40`, where it stays even after 50 epochs.

The failure of this experiment shows that the dataset employed is complex and only one GRU unit is not enough to generalize it.

5.7.2 Experiment 2: Using two Bi-GRUs

By incorporating two bidirectional GRUs, this experiment provides an implementation that is structurally closer to the LipNet model.

This architecture is able to generalize the narrowed GRID dataset and as such it became the architecture that gets covered in-depth in chapters 4 and 6 of this work, which is why we are not going to focus on it here.

Briefly, the model achieves a minimum average training loss of `0.1798` and average validation loss of `0.0748`, which allows it to obtain an accuracy of 88.04% on the testing dataset.

5.7.3 Experiment 3: Using two bidirectional LSTMs

Long short-term memory units, also known as LSTMs, are older yet more complex than GRUs, as addressed earlier in chapter 4. The idea of this experiment was that one might obtain better results by utilizing LSTMs as opposed to GRUs.

The experiment was also prompted by Renotte's LipNet implementation [9], which differs from Assael's approach in that it uses bidirectional LSTMs as opposed to bidirectional GRUs.

The training of this experiment was entertained for around 50 epochs. After the average training loss did not decrease under `1.05`, as it can be observed in figure 28, the decision to halt the experiment was called and the author decided that the model to be used for testing is the one trained during experiment 2.

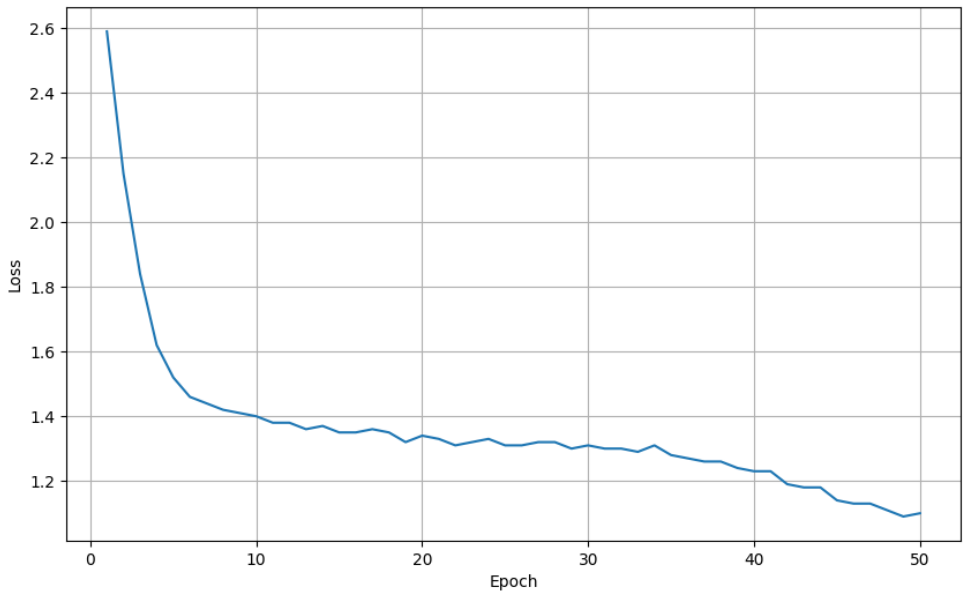

Figure 28: Training loss for the first 50 epochs of experiment 3 with a recurrent phase architecture consisting of two bidirectional LSTMs

The average training losses of the three experiments, over the first 50 epochs, are compared in figure 29.

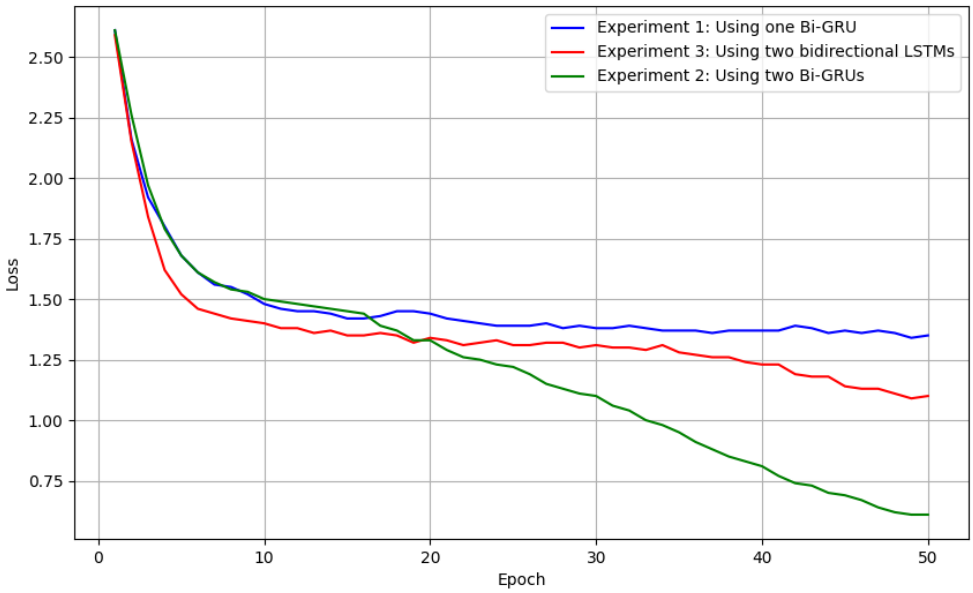

Figure 29: Graph of the average training loss for the three experimental models over the first 50 epochs

5.8 Computing CTC loss correctly

During the development of the *LipReaderTrainer* training script, calculating the value of CTC loss correctly proved tricky.

A correct implementation can be checked in listing 5.6 and was achieved only after consulting the PyTorch documentation[1], and involves permuting the data and calculating certain sizes correctly.

```
frame_lengths = torch.full((frames.size(0),), outputs.size(2), dtype
   =torch.long)
alignment_lengths = torch.tensor([len(alignment) for alignment in
   alignments], dtype=torch.long)
train_loss = self.criterion(outputs.log_softmax(2).permute(1, 0, 2),
    alignments, frame_lengths, alignment_lengths)
```

Listing 5.6: CTC loss correct calculation, excerpt taken from LipReaderTrainer class

5.9 Implementing the character converter

As explained in chapter 4, a class is needed to define the vocabulary allowed for predictions and convert a text containing characters from the vocabulary to indices and vice versa. *CharConverter* is modelled based on `tensorflow.keras.layers.StringLookup`, which is a TensorFlow-specific implementation of converting chars to numbers from a vocabulary given as argument. And since there is no equivalent in PyTorch, it had to be coded by the author.

The implementation follows the useful functionalities of `StringLookup` almost 1:1, is robust, generic and a point of pride for the author.

Attached in listing 5.7 is *CharConverter*'s method `convert_char_to_idx()`, which can take a variety of inputs and is used during the data processing phase for alignments.

```
from typing import List, Union

def convert_char_to_idx(self, char: Union[str, List[str], bytes,
   List[bytes]]) -> torch.Tensor:
```

[1] https://pytorch.org/docs/stable/generated/torch.nn.CTCLoss.html

```
    if isinstance(char, bytes):
        return torch.tensor(self.char_to_idx[char], dtype=torch.
           int32)
    if isinstance(char, str):
        return torch.tensor(self.char_to_idx[char.encode()], dtype=
           torch.int32)
    # Convert a list of characters to the list of their
       corresponding indices
    idx = []
    for character in char:
        if isinstance(character, bytes):
            idx.append(self.char_to_idx[character])
        elif isinstance(character, str):
            idx.append(self.char_to_idx[character.encode()])
    return torch.tensor(idx, dtype=torch.int32)
```

Listing 5.7: CharConverter's `convert_char_to_idx()` method implementation

5.10 Correcting and interpreting the results

When finally obtaining a model that could lip read decently enough on the narrowed GRID dataset, the challenge became to interpret its accuracy. For this, the word error rate (WER), which measures how many words the model predicts correctly, was the main standard to measure LipLink's performance and compare it to other already-existing models.

However, the author found this methodology to not be entirely accurate of the results obtained, as LipLink's predictions can be accompanied by a lot of noise, which can lead to a high WER, even though the predictions are often correct.

As such, one crucial task for the success of this work was to look for other ways to measure the quality of the outputs. Drawing upon the work of Kaleb Shah[2], which also implements a simplified approach to LipNet, a decision was taken to use the edit distance, measuring the number of modifications to a sentence, as the prime method of measuring the model's accuracy.

Another need was to reduce the noise that the model's predictions often contain, and this was based on Python libraries that specifically remove typos in a text, such as `textblob` or `python-spellchecker`.

[2]Shah, Kaleb, *Recreating LipNet: A Simplified Approach to Lip Reading Neural Networks*

5.11 Type hinting

Working on production code in Python opened the author's eyes to type hinting, which is a mechanism for Python to describe the type of the arguments a method takes, as well as a method's return type. Such type hinting is exemplified in listing 5.7.

The implementation of type hinting is provided in the default Python library `typing` and defines classes such as `List`, which indicates that the provided structure is a list, and `Union`, which indicates that the type of a variable is one of several types mentioned inside square brackets.

Type hinting is a great addition to a traditionally dynamically-typed language and sticking to it was, in the opinion of the author, both a great challenge and something to abide by.

5.12 Virtual environment

Python virtual environments are a much-needed way to control the distribution and versioning of Python packages and not crowd the system Python. For users that use Python for many different problems, it is a breath of fresh air and, in the author's opinion, a must for maintaining a Python codebase clean.

The virtual environment, usually shorted to `venv`, works by creating a directory containing a copy of a Python version on which to install the desired packages for a project. The virtual environment can be activated and deactivated by running a file in the directory.

To provide the package version used so as to increase reproductibility of a result or a bug, one can store a list of packages on a file usually called `requirements.txt` that can be simply installed by another user.

Considering the current project used a large array of packages, the author believes that the providing of a `requirements.txt` file is a must for meeting a certain codebase quality standard.

6 RESULTS

This chapter aims to capture the results obtained by testing LipLink, and providing the readers with both a qualitative and a quantitative picture of where the model is situated in its relationship to (1) its initial goals, as well as (2) other already-existing automated lip readers.

The model whose results we are going to analyze is the one described in section 5.7.2, and whose architecture has been explained at large in chapter 4. Briefly, it is a complex neural network that is able to generalize the narrowed GRID dataset and obtain an accuracy, in the sense of *Levenshtein distance* which will be thoroughly explained in what follows, of 88.04%.

The following sections cover LipLink's results and splits them into qualitative and quantitative results. The quantitative analysis is broader, as it is usually the case for machine learning papers, but it is the solemn belief of the author that the qualitative analysis presents us with more valuable and harder to overlook results.

6.1 Qualitative analysis

LipLink is a success, and the reason it is a success is that it achieved all the goals that were stated in its inception. As described at large in chapter 2, the goal of LipLink was to provide a simplified implementation based around Yannis M. Assael's LipNet [1], that is able to lip read with a good enough accuracy on a well-known dataset, all while utilizing less computational resources.

The fact of the matter is that, the description of the qualitative goal of this research is exactly what LipLink is. With just a fraction of the GRID dataset and a simplified code structure, LipLink can consistently predict what is being said in a video based only on the movement of the lips.

On top of that, LipLink's implementation provided the author with valuable experience, which the detailed description that was given in chapter 4 can certainly attest to.

Beyond the measurable results that will be detailed in what follows, LipLink is an application that *works correctly* and that provides us with an ingenious and simple proof-of-concept way of implementing automated lip reading, which can be trained even on a personal computer.

6.2 Quantitative analysis

6.2.1 Training and validation loss

During training, the average loss calculated on the training dataset, as well as on the testing one, decreases as one would expect. Compared to the other experiments, the model passes the CTC loss threshold of around `0.70` and starts predicting most words in the sentence around epoch 45.

At epoch 100, the model achieves an average training loss of `0.1798` and an average validation loss of `0.0748`, which are the global minimums for the entire training phase.

The graph of the training and testing loss for the first 100 epochs of the winning experiment can be checked in figure 30.

For measuring the model's accuracy, we propose two different approaches: the word error rate and the Levenshtein distance, and explain their advantages, disadvantages and how the model relates to the already existing ones.

6.2.2 Word error rate

The word error rate (WER), is by far the most popular method of measuring the accuracy of automatic speech recognition and automatic lip reading solutions. It involves counting how many words the model failed to predict from the total amount of words that were said.

For example, if the target was `bin red by t two please`, and the prediction is `set green by t two please`, the WER would be calculated as `2 / 6`, or 33.33%, with the model failing to predict the words `bin` and `red`.

After post-processing the data and removing the typos and the noise that the model generated, our model achieves a WER of 20.5% on the entire testing dataset, with an average 1.23 wrongly predicted words for every sentence out of the standard 6.

As can be observed in figure 31, LipLink performs significantly worse than the already existing architectures on predicting the words when trained on the GRID dataset, with Assael's LipNet achieving a 4.6% WER on the dataset and Pingchuan's VSR a stunning 1.2%.

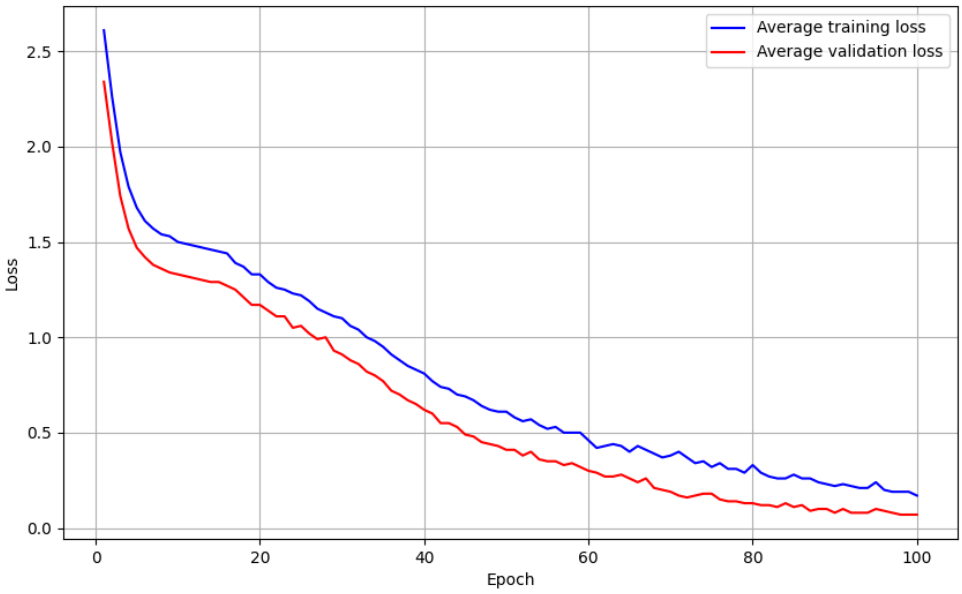

Figure 30: LipLink average training and validation loss for the first 100 epochs

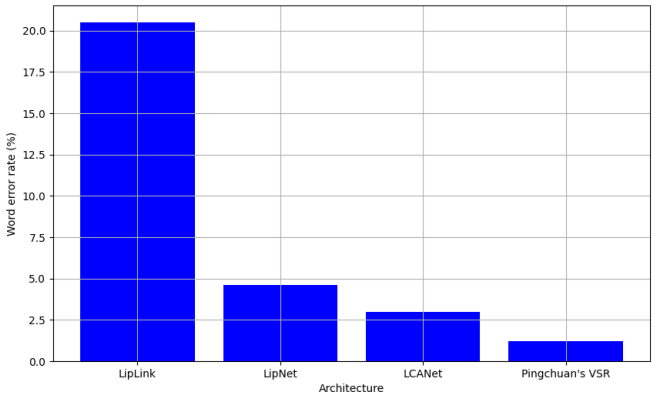

Figure 31: Bar chart comparing the LipLink's WER to other state-of-the-art implementations discussed at large in chapter 2 of this paper[1]

However, this statistical analysis is very misleading. First of all, LipLink was trained on a tiny subset of the GRID dataset, which was discussed previously as *narrowed GRID*. As stated previously, the purpose of LipLink is to serve as proof-of-concept of what can be achieved even with limited computing capabilities.

Compared to this, the aforementioned state of the art models are trained on the entire GRID dataset, or, as is the case of Pingchuan's VSR [4], on large datasets containing videos with automatically generated text as labels. This is exemplified in figure 32.

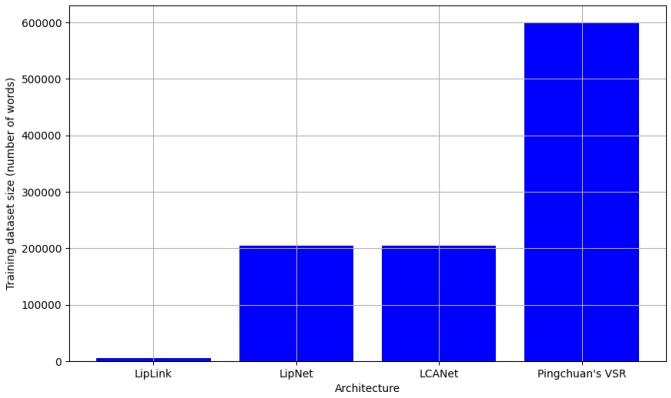

Figure 32: Bar chart comparing the LipLink's dataset size to other state-of-the-art implementations discussed at large in chapter 2 of this paper

LipLink serves, as intended to from the very beginning, as a simplified approach to automated lip reading, the *optimal* point between accuracy and simplicity: too simple a model would perform very badly, while too complicated a model would greatly impact the application's repeatability and reach.

[1] Lipreading on GRID corpus (mixed-speech). https://paperswithcode.com/sota/lipreading-on-grid-corpus-mixed-speech

Second of all, many typos that are not eliminated by the correcting phase are marked as wrong predictions, when in reality the model predicted them correctly. An example of this is the prediction for the target `bin red by t two please`, which the model predicts as `bin reed by t two please`, however because the model predicted character `e` for more than one frame and because `reed` is an actual word and therefore not eliminated by the correcting function, it counts as a word error and increases the WER of the sentence from a perfect 0% to 16.67%.

Lastly, the words that the model predicts with the highest WER are mostly 1-character words, that appear in every sentence from the GRID dataset. For example, the predictions for targets `set green in v zero now` as `set green in z zero now` and `lay green at l nine soon` as `lay green at a nine soon` are, in the opinion of the author, good predictions, however they both have a WER of 16.67% because of the model failing to predict the 1-character word.

This is why, in the opinion of the author, the better way to measure accuracy, especially in the case of LipLink, is not the word error rate, but actually the edit distance.

6.2.3 Levenshtein distance

An edit distance is any function that measures the number of *edits* or modifications needed to perform on a string so that it becomes the second string. One of the most common edit distance functions in use today is the Levenshtein distance. Today, it is used for all kinds of tasks, including, but not limited to, DNA analysis and spell checking.

We are going to use it to calculate the lip reading accuracy of LipLink at the sentence level. For example, for the target `bin red by t two please` and the prediction `bin reed by t two please`, which have a WER of 16.67%, the Levenshtein distance is 1, as can be deduced from figure 33, and the Levenshtein-based accuracy would compute as `1 / 24`, or just 4.16%.

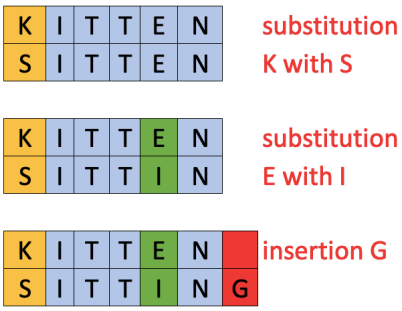

Figure 33: Example[2] of two strings having a Levenshtein distance of 3

[2]© https://miro.medium.com/v2/resize:fit:1298/1*8CkgO-S3Oc4WQuPtYPz59Q.png

The Levenshtein distance has therefore been used to measure the performance of LipLink on the testing dataset of 100 sentences, and the Levenshtein distance was found to be 11.96% on the entire dataset, or just 2.98 per sentence. Therefore, the accuracy of the LipLink model has been given as the opposite of the Levenshtein distance, or 88.04%. As mentioned in section 6.2.2, most of these wrongly-placed characters are noise generated by the model that humans can easily overlook.

Keeping in mind that all sentences in the testing dataset have exactly 6 words and an average of 24.92 characters, this leads us to the summary of quantifiable results expressed in table 4.

Table 4: Summary of quantifiable results of LipLink on the testing dataset

Metric	**LipLink's performance**
Word error rate, total	20.5%
Word error rate, words per sentence	1.23
Levenshtein distance, total	11.96%
Levenshtein distance, characters per sentence	2.98
Accuracy, total	**88.04%**

The quantitative results are, in the opinion of the author, satisfying, and do an extraordinary final job at painting the picture of what LipLink *really* is: a reliable, solid solution of automated lip reading trained with just a fraction of what other state of the art models were (see figure 32).

For reference, the author wants to divert your attention to table 5, which showcases how the model learns to predict a sentence on which it performs at the end with maximal accuracy.

Table 5: Predictions for a sentence from the validation dataset at different moments during the training process

Epoch	**Prediction for target `lay green at z seven soon`**
10	`sit rlee by five soaawn saolaiinn ien lw`
20	`lay rluee ait t fine nooawn awi w ii`
30	`lay reee ay t fine soonnnw i in`
40	`lay greed ay t sive soonn inninnnwnnw`
50	`lay green in t sivee soonin ilnppeegrenw`
60	`lay green at f siven soonn iinppeegenw`
70	`lay green at p seven soon wedppplereenwwit`
80	`lay green by j seven soonwgeennin`
90	`lay green by s seven soonwhen`
100	`lay green at z seven soon`

7 CONCLUSION

LipLink is a machine learning based application that is able to interpret what is being said solely based on the movement of the lips, achieving an accuracy of 88.04%, as already shown at large above.

This paper follows the journey of implementing LipLink, paying close attention to its goals, similar state of the art architecture, its class and neural network model architecture, the problems encountered during its development and the proposed solutions, as well as its performance on well-established datasets and its relation in this context to other already-existing implementations.

Concerning its relationship to other models, the LipLink architecture is profoundly inspired by the architecture described in Yannis M. Assael's 2016 seminal paper on automated lip reading, LipNet, and attempts to implement a similar PyTorch-based architecture that is significantly *simpler* and can reach a *wider* range of less sophisticated systems, in a manner that is similar yet decidedly distinct to Nicholas Renotte's TensorFlow-based implementation.

In this regard, the object oriented implementation of LipLink, shown in figure 13, employs a wide arrange of classes tasked with pipelining and preprocessing data, defining the neural network model and training it, as well as correcting the outputs and testing the application's accuracy. In order to be able to generalize the small dataset with such accuracy, the neural network model that is behind LipLink, portrayed as in figure 21, is *very complex*, containing a convolutional, recurrent and fully-connected phases that contain around 1.4 million trainable parameters, which get trained to a minimal loss of 0.0748, as shown in figure 30.

To contribute to the stated goal of increased usability of this current work for beginners in the field of machine learning, the paper then closely details some of the problems that appeared during LipLink's developmental process, highlighting how they can be solved efficiently. On top of that, up to three experiments with different model architectures have been undertaken, the results of which, as well as their comparison, have been provided in the paper as shown in figure 29.

Afterwards, LipLink's results are analyzed both quantitatively and qualitatively, and these results serve to solidify LipLink's position in the context of the other state of the art solutions that were discussed. On top of all this, LipLink has achieved its stated goal of providing users with a simplified approach to sentence-level lip reading that could serve the basis for a more complex approach that could be integrated in smart devices, with the goal of benefiting the lives of hearing-impaired people.

Some of the methods that might be used with the purpose of further development of this work include the implementation of an Attention mechanism, such as those used in LCANet or Pingchuan's VSR, the further enlargement of the dataset with the aimed purpose of maximizing the data variety, while preserving a high-enough quality, as well as looking to improve the choice of hyperparameters, which are all methods of improvement that were recognized during the incipient phase of researching state of the art models.

As a final conclusion, building LipLink was a very engaging experience that challenged the author's ability to research, understand, develop and debug machine learning models. There is no doubt that the experience gathered will serve well at the engineering of even more complex systems.

BIBLIOGRAPHY

[1] Assael Yannis et al. Lipnet: End-to-end sentence-level lipreading. *Computing Research Repository (CoRR)*, abs(1611):01599, 2016.

[2] Kai Xu et al. Lcanet: End-to-end lipreading with cascaded attention-ctc. *Computing Research Repository (CoRR)*, abs(1803):04988, 2018.

[3] M. Cooke et al. An audio-visual corpus for speech perception and automatic speech recognition. *The Journal of the Acoustical Society of America*, 120(5):2421–2424, 2006.

[4] Pingchuan Ma et al. Visual speech recognition for multiple languages in the wild. *Computing Research Repository (CoRR)*, abs(2202):13084, 2022.

[5] Yann LeCun et al. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. Last accessed: June 16, 2024.

[6] Bryce Hall. The state of ai in 2022—and a half decade in review. `https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai-in-2022-and-a-half-decade-in-review/`. Last accessed: June 16, 2024.

[7] Yoshua Bengio Ian Goodfellow and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, Massachusetts, 2016.

[8] Ryan O'Connor. Pytorch vs tensorflow in 2023. `https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/`. Last accessed: June 19, 2024.

[9] Nicholas Renotte. Build a deep learning model that can lip read using python and tensorflow. `https://www.youtube.com/watch?v=uKyojQjbx4c`. Last accessed: June 18, 2024.